Ruprecht-Karls-Universität
Heidelberg
Hochschule Heilbronn

Diplomstudiengang
Medizinische Informatik

Diplomarbeit

Entwicklung einer performanten Volume Rendering Komponente für das Operationsplanungssystem MOPS 3D

Sascha Diatschuk

Januar 2012

Referent: Prof. Dr.-Ing. Hartmut Dickhaus
Korreferent: Dr. med., Dipl. Phys., M. Sc. Roland Metzner
Betreuer: Dipl.-Inform. Med. Urs Eisenmann

Danksagung

Bedanken möchte ich mich bei Herrn Prof. Dr.-Ing. Hartmut Dickhaus für die Ausgabe des Themas und dafür, dass ich die Diplomarbeit unter solch angenehmen Arbeitsbedingungen im Institut für Medizinische Biometrie und Informatik anfertigen konnte.

Weiterer Dank gilt Herrn Dr. med., Dipl. Phys., M. Sc. Roland Metzner für die Übernahme des Korreferats, für die Beratung in medizinischen Fragen und für die Bereitstellung medizinischer Datensätze.

Ganz besonderer Dank gebührt Herrn Dipl.-Inform. Med. Urs Eisenmann für die Betreuung meiner Arbeit, für den Entwurf des Themas und für die gute Beratung bei Schwierigkeiten hinsichtlich der Programmierung.

Bei meinem Kollegen und Mitbewohner Marius Wirths möchte ich mich ebenfalls bedanken für die gute Zusammenarbeit an der GPU-Segmentierung.

Ich bedanke mich weiterhin bei Alexander Bock und Timo Ropinski vom Voreen Entwicklungsteam, die mir bei meinem Problem mit dem Combine Prozessor geholfen haben.

Zu guter Letzt bedanke ich mich bei meinen Eltern und Großeltern, die mir mein Studium ermöglicht haben, bei meiner Freundin, die mich immer auf Trab hält, motiviert und unterstützt, bei meinem Bruder für seine musikalische Untermalung während der Arbeit und bei meinen Mitbewohnern, bei denen ich mich immer heimisch fühle.

Zusammenfassung

Computergestützte Planung von chirurgischen Eingriffen gehört in der heutigen Zeit zum klinischen Alltag. Besonders in der Neurochirurgie ist ein hohes Maß an Präzision erforderlich, so dass es hilfreich ist, die räumlichen Gegebenheiten des individuellen Gehirns, die für die Operation von Bedeutung sind, schon im Voraus zu kennen.
MOPS 3D ist ein Operationsplanungssystem, das imstande ist, verschiedene medizinische Bilddaten einzulesen und dem Chirurgen eine dreidimensionale Darstellung des Körperinneren zu präsentieren. Statt mit der herkömmlichen Oberflächendarstellung kann dieses seit Kurzem auch mit Hilfe von Volumenvisualisierung (Volume Rendering) dargestellt werden, da das Volume Rendering Framework *Voreen* prototypisch in MOPS 3D integriert wurde.
In der nachfolgenden Arbeit wird diese prototypische Integration auf Schwachstellen im Verhalten und auf Verbesserungspotential von Performanz und Stabilität untersucht. Danach werden entsprechende Verbesserungen und funktionale Neuerungen konzipiert und in der Volume Rendering Komponente implementiert.
Die Integration der neuesten Version von Voreen nimmt dabei einen wichtigen Platz ein, da dieses Framework seit der letzten Integration viele grundlegende Änderungen erfahren hat und deshalb ein Gewinn an Performanz zu erwarten ist.
Bei den neuen Funktionen ist die Beschneidungsfunktion von Volume Rendering Segmenten von großer Bedeutung. Diese werden normalerweise mit Hilfe eines Quaders beschnitten, so dass nur der Teil eines Segments dargestellt wird, der sich im Quader befindet. Es wird eine neue Variante konzipiert und implementiert, die die Anwendung beliebiger Begrenzungsstrukturen ermöglicht. Dies wird anhand eines Ellipsoids demonstriert.

Weiterhin wird die Beschneidungsfunktion umgestaltet, dass sie auch negativ anwendbar ist, d.h. nur Teile des Segments darstellt, die außerhalb der Begrenzungsstruktur liegen. Eine wichtige neue Funktionalität ist die Integration eines Segmentierungsverfahrens, das auf der Grafikkarte ausgeführt wird. Dieses Verfahren wird zeitgleich zu dieser Arbeit von Marius Wirths im Rahmen seiner Diplomarbeit ausgearbeitet. Bei der Integration wird eine geeignete Kommunikationsschnittstelle und eine intuitive Benutzerschnittstelle zur Steuerung der Segmentierung entwickelt.

Eine abschließende Evaluierung zeigt, dass der Performanzgewinn durch das Upgrade auf die neueste Voreen Version selbst auf einer mittelmäßigen Grafikkarte enorm ist und die Erwartungen übertrifft. Weiterhin stellt die neue Beschneidungsfunktion eine sinnvolle und flexible Ergänzung dar, wenngleich die Darstellungsgeschwindigkeit etwas niedrig ist, verglichen mit dem bereits integrierten Begrenzungsverfahren für Quader.

Die Benutzeroberfläche zur Steuerung des Segmentierungsverfahrens erweist sich generell als sehr nützlich, um schnell geeignete Ergebnisse zu erzielen und stellt eine solide Grundlage dar, die viel Platz für Erweiterungen und Weiterentwicklung auf dem Gebiet der Segmentierung bietet.

Insgesamt zeigt sich, dass ein wichtiger Schritt von der prototypischen Integration zu einer klinisch einsetzbaren Anwendung getan wurde und eine performante Volume Rendering Komponente entstanden ist.

Inhaltsverzeichnis

1 Einleitung

In der Neurochirurgie ist heutzutage der Einsatz von Operationsplanungssoftware vor und während der Operation an der Tagesordnung. Aufgrund der filigranen Strukturen des Gehirns und des hohen Verletzungsrisikos für den Patienten, verlassen sich Chirurgen auf computergestützte Verfahren, die ein zusätzliches Maß an Genauigkeit bieten. Diese Verfahren basieren meist auf Daten der medizinischen Bildgebung (üblicherweise aus der Computer- oder Kernspintomographie) und visualisieren diese Daten dreidimensional für den Operateur.

Hierbei ist das Ziel einer Operationsplanung hauptsächlich die Darstellung von wichtigen Strukturen in der zu operierenden Körperregion, wie z.B. die Abgrenzung eines Tumors zur umliegenden Umgebung. Üblicherweise wird bei solchen Strukturen mit einem Segmentierungsverfahren ermittelt, welche Teile der Bilddaten zur Struktur gehören. Erst anschließend kann sie dreidimensional visualisiert werden, da dem Programm bekannt sein muss, was dargestellt werden soll. Die Darstellung in 3D erfolgt dann meist durch die Berechnung der Oberfläche der segmentierten Region und dem nachfolgenden Rendern[1] dieser Oberfläche. Man spricht hierbei passend von Oberflächendarstellung oder Surface Rendering.

Im Gegensatz dazu ist in den letzten Jahren ein weiteres Visualisierungsverfahren zunehmend interessant geworden, das in der Lage ist, die gesamten Bilddaten eines Volumens zur Darstellung zu nutzen, statt nur seiner Oberfläche. Dieses Verfahren heißt Volume Rendering (Volumendarstellung). Der Vorteil gegenüber der Oberflächendarstellung be-

[1] Rendering beschreibt den Prozess der Projektion einer dreidimensionalen Punktmenge auf eine zweidimensionale. Da in der Fachliteratur fast ausschließlich dieser englische Begriff Verwendung findet, wird in dieser Arbeit ebenfalls Gebrauch vom eingedeutschten Wort "rendern"gemacht.

steht darin, dass auch das Innere einer Struktur visualisiert werden kann. Das Konzept, das hinter diesem Verfahren steckt, gibt es schon seit langer Zeit (es trat erstmals in einer Veröffentlichung von 1988 auf [Sab88]). Da es allerdings sehr rechenintensiv ist, konnte es sich erst in den letzten Jahren steigender Beliebtheit erfreuen mit der Erschließung der Grafikkartenprogrammierung. Der Hauptprozessor (CPU) eines Computers ist für die sequentielle Abarbeitung genereller Berechnungen ausgelegt. Ein Grafikkartenprozessor (GPU) dagegen ist auf 3D Berechnungen spezialisiert und kann mehrere Berechnungen parallel ausführen, weshalb er um ein Vielfaches schneller arbeiten kann als ein Hauptprozessor. Durch das Auslagern des Volume Rendering auf den Grafikkartenprozessor sind heutzutage selbst mittelmäßige Rechner in der Lage, Volume Rendering flüssig anzuwenden.

1.1 Motivation

Dass Volume Rendering keine Hochleistungsrechner benötigt, um medizinische Bilddaten darzustellen, wird ersichtlich im neurochirurgischen Operationsplanungssystem MOPS 3D [Eis+02]. Diese Anwendung sollte um die Möglichkeit zur Volumenvisualisierung erweitert werden. Aus diesem Grund evaluierte Andreas Freudling im Rahmen seiner Diplomarbeit [Fre09] verschiedene Volume Rendering Frameworks für eine mögliche Nutzung in MOPS 3D. Voreen [MS+09b] erzielte dabei das beste Ergebnis, woraufhin er dieses Framework prototypisch integrierte. Dazu entwickelte er eine Benutzerschnittstelle, die es ermöglicht, die in MOPS 3D geladenen MRT[2] Bilddaten mit Volume Rendering darzustellen und grundlegende Interaktionen mit der Volumendarstellung durchzuführen. In dieser Arbeit wurde deutlich gemacht, dass es möglich ist, Echtzeit Volume Rendering in MOPS 3D zu integrieren und zu nutzen.
Das Problem bei dieser prototypischen Fassung der Volume Rendering Komponente ist, dass sie einige Schwächen aufweist hinsichtlich der Performanz und Stabilität. Des Weiteren gibt es zahlreiche sinnvolle Funktionen, um die sich die Volume Rendering

[2]Magnetresonanztomograph.

Komponente erweitern ließe, um den klinischen Nutzen zu erhöhen und die Benutzerfreundlichkeit zu steigern.

1.2 Zielsetzung

Aufgrund des prototypischen Charakters der bisherigen Integration von Voreen sollen dort Schwächen aufgezeigt und entsprechende Verbesserungsvorschläge konzipiert werden. Weiterhin soll analysiert werden, wie die Funktionalität der Volume Rendering Interaktion sinnvoll erweitert werden kann. Das Hauptziel hierbei ist es, die Volume Rendering Komponente von MOPS 3D so zu verbessern, dass sie einen größeren Nutzen in der klinischen Anwendung bietet.
Ein wichtiger Schritt, um die Performanz und Stabilität zu steigern, soll die Integration der neuesten Voreen Version 2.6.1[3] in MOPS 3D sein, da sie im Vergleich zur aktuell integrierten Version r13[4] grundlegende Veränderungen mit sich bringt und eine potentielle Verbesserung der Darstellungsgeschwindigkeit bewirkt.
Des Weiteren sollen funktionale Neuerungen entwickelt werden, die das Verfahren der Beschneidung von Volume Rendering Segmenten erweitern. Dieses ist bisher auf einen Quader als Begrenzungsstruktur beschränkt. Es soll möglich sein, jede beliebige geometrische Struktur zur Begrenzung hinzuzufügen. Außerdem soll der Benutzer in der Lage sein, den Beschneidungsbereich bei Bedarf zu invertieren, so dass alles innerhalb des Bereichs ausgeblendet wird – im Gegensatz zum normalen Verhalten, das alles außerhalb des Bereichs ausblendet.
Eine zusätzliche wichtige Neuerung soll überdies die Integration eines GPU-basierten Segmentierungsverfahrens sein, welches von Marius Wirths zeitgleich im Rahmen seiner Diplomarbeit [Wir12] entwickelt wird. Ein besonderes Augenmerk soll dabei zum einen auf dem Entwurf einer internen Codeschnittstelle liegen, die mit diesem Segmentierungsverfahren kommuniziert, und zum anderen auf der Gestaltung einer grafischen Benutzeroberfläche, die intuitive Benutzerinteraktion ermöglicht.

[3] Veröffentlichung am 4. November 2010.
[4] Veröffentlichung am 13. Februar 2009.

Die oben genannten Verbesserungen und Neuerungen sollen anschließend implementiert und evaluiert werden.

2 Grundlagen

2.1 Volume Rendering

Der Begriff Volume Rendering bezeichnet das Verfahren der direkten Volumenvisualisierung, das nicht wie bei der indirekten Volumenvisualisierung eine zuvor berechnete Repräsentation des Volumendatensatzes darstellt, sondern direkt auf den einzelnen Voxeln[5] des Datensatzes operiert.
Der Hauptvorteil des Volume Rendering im Gegensatz zur herkömmlichen Darstellung der Oberflächen (*Surface Rendering*) liegt darin, dass Volume Rendering durch Transparenz Einblicke in das Volumen ermöglicht, da auch der Inhalt bei der Darstellung berücksichtigt wird – nicht nur die Hülle, wie bei der Oberflächendarstellung. Für den medizinischen Anwender ist das sehr nützlich, weil für ihn auch die innere Beschaffenheit von Strukturen interessant ist. Ein weiterer Vorteil liegt darin, dass Volume Rendering sofort berechnet wird, ohne vorher die Oberflächen einer Struktur zu ermitteln, wie das normalerweise der Fall ist. Auf die Wichtigkeit dieses Vorteils wird in Kapitel 3.3 näher eingegangen.

2.1.1 Mathematische Grundlagen

Für das weitere Verständnis von Volume Rendering ist das mathematische Modell, welches Volume Rendering zu Grunde liegt, von Bedeutung.
Rendering an sich lässt sich reduzieren auf ein Problem von Interaktion zwischen Lich-

[5] Ein Voxel ist das dreidimensionale Äquivalent eines Pixels (zusammengesetzt aus „**Vo**lume Pi**xel**“), entspricht also einem bestimmten Punkt in einem Volumen.

tern, Objekten und dem Medium dazwischen. Das Dichteemittermodell von Sabella [Sab88] nutzt diesen Ansatz, um jedes Partikel des Volumens als eigenständige Lichtquelle (Emitter) zu betrachten, dessen Licht auf seiner Reise durch das Volumen teilweise oder vollständig absorbiert wird. Zur Berechnung des Lichts wird ein Strahl durch das Volumen zum Blickpunkt geschickt und die einzelnen Intensitäten auf jeder Position des Strahls akkumuliert (man spricht vom sog. *Raycasting*). Abbildung 1 verdeutlicht dieses Verfahren.

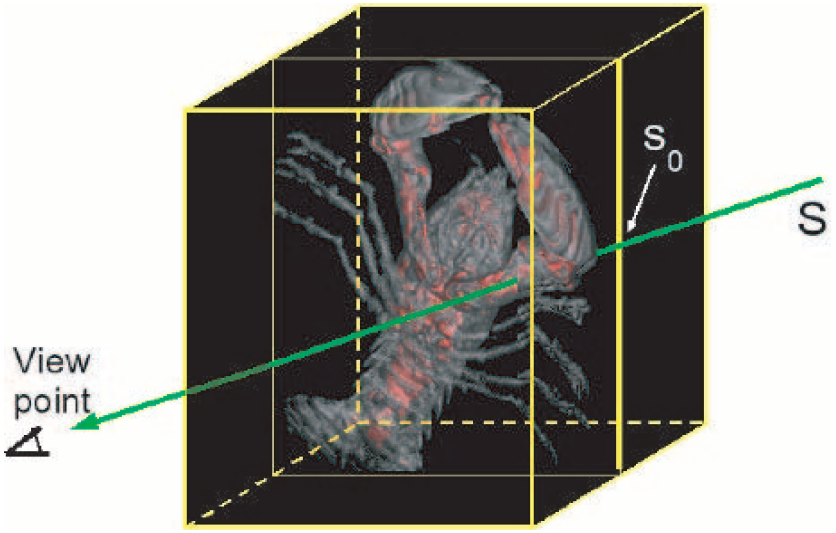


Abbildung 1: Ein Strahl durchquert das Volumen vom Einstiegspunkt S_0 zum Blickpunkt

Quelle: [PB07, S. 185]

Die folgende Differentialgleichung dient zur Berechnung der Intensität:

$$\frac{dI}{ds} = Q(s) - \tau(s) \cdot I(s) \tag{2.1}$$

$I(s)$ ist dabei die Intensität des Lichts an der Stelle s des Strahls. $Q(s)$ ist der Beitrag des Volumens zum finalen Bild. Dieser wird maßgeblich durch die sog. Transferfunktion beeinflusst (siehe Kapitel 2.1.3). Der Absorptionskoeffizient $\tau(s)$ schwächt die Intensität entlang des Strahls ab. Löst man diese Differentialgleichung so ergibt sie die sog. Volume

Rendering Gleichung:

$$I(s) = I_{s_0} \cdot e^{(-\int_{s_0}^{s} \tau(t)dt)} + \int_{s_0}^{s} Q(p) \cdot e^{(-\int_{p}^{s} \tau(t)dt)} dp \tag{2.2}$$

I_{s_0} ist dabei der initiale Intensitätswert beim Eintritt in das Volumen. Er wird auch das Hintergrund-Umgebungslicht genannt.
Da die Gleichung nicht für alle Transferfunktionen und Interpolationen analytisch gelöst werden kann, muss sie mit Hilfe des Riemannschen Integrals numerisch gelöst werden. Dabei muss eine konstante Schrittgröße Δs festgelegt werden und die Volume Rendering Gleichung muss diskretisiert werden. Die daraus resultierende Gleichung lautet:

$$I(s) = I_0 \prod_{k=0}^{n-1} t_k + \sum_{k=0}^{n-1} Q(k \cdot \Delta s) \cdot \Delta s \prod_{j=k+1}^{n-1} t_j \tag{2.3}$$

Dabei gilt $n-1$ ist die letzte Position des Strahls und $k = 0$ ist der Eintrittspunkt. t_k ist die Transparenz an der Stelle k. Diese Gleichung tastet das Volumen also entlang des Strahls ab und akkumuliert die Beiträge an den diskreten Stellen k [vgl. PB07, S. 183ff.]

2.1.2 Operationen

Um die Volume Rendering Gleichung in der Praxis anzuwenden sind eine Reihe von Operationen nötig, die bei vielen Volume Rendering Implementierungen im Grunde gleich sind – mit Ausnahme der Reihenfolge, in der sie ausgeführt werden. Im Folgenden werden die einzelnen Operationen in der Reihenfolge erklärt, in der sie in einer gängigen Volume Rendering Routine namens Post-Klassifikation[6] vorkommen [vgl. PB07, S. 187ff.].

- **Abtasten**: Das Volumen wird entlang des Strahls nach dem Shannon'schen Abtasttheorem abgetastet. Werte, die zwischen einzelnen Voxeln liegen werden dabei (meist trilinear) interpoliert.

[6] Die Namensgebung rührt von der Reihenfolge der Operationen, da die Klassifikation nach dem Abtasten stattfindet.

- **Klassifikation und Beleuchtung**: Mit Hilfe der Transferfunktionen (siehe Kapitel 2.1.3) wird der abgetastete Punkt klassifiziert, d.h. ihm wird eine Farbe und Opazität – die Deckkraft (als Gegenteil der Transparenz zu verstehen) – zugeordnet. Mit Hilfe dieser Werte wird anschließend die Beleuchtung des abgetasteten Punktes berechnet (auch *Shading* genannt).

- **Komposition**: Die abgetasteten Werte werden akkumuliert. Gängige Kompositionsverfahren sind die Addition aller Werte oder die Suche nach dem Abtastwert mit der höchsten Intensität (die sog. *maximum intensity projection*).
  Am häufigsten wird das additive Verfahren angewandt. Dort kann der Strahl abgebrochen werden, sobald die addierten Werte eine bestimmte Opazität (z.B. 100%) übersteigen, was einen wichtigen Schritt zur Performanz beiträgt.

2.1.3 Transferfunktionen

Transferfunktionen ordnen einem abgetasteten Wert oder Voxel (bzw. Bereichen von Werten) aufgrund seiner Eigenschaften (z.B. Intensität oder Gradient) eine Farbe und Opazität zu. Durch diese Zuordnung wird ein Punkt klassifiziert, d.h. ein Voxel wird anhand seines Grauwertes einer bestimmten Körperstruktur zugewiesen.
Die einfachste Methode bilden die Histogramm-Transferfunktionen. Das Grauwerthistogramm des Volumens wird berechnet und einzelnen Bereichen wird eine Farbe und Opazität zugeordnet. Abbildung 2 zeigt ein Beispiel für ein Histogramm mit eindimensionaler Transferfunktion. Auf der x-Achse sind die Grauwerte eingetragen und auf der y-Achse die Häufigkeiten. Dem Grauwertbereich oberhalb eines Farbbalkens wird jeweils die darunter liegende Farbe zugeordnet.

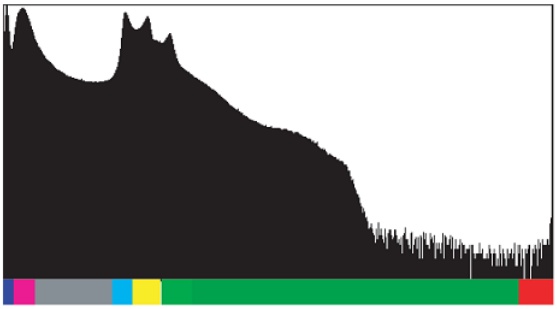


Abbildung 2: Grauwerthistogramm mit eindimensionaler Transferfunktion
Quelle: [PB07, S. 141]

Mit Hilfe einer zweidimensionalen Transferfunktion lässt sich auch die Opazität zuordnen. Die Zuordnung erfolgt dabei derart, dass ein Grauwert immer die Opazität bekommt, die dem y-Wert der Kurve an seinem x-Wert entspricht. Abbildung 3 zeigt ein Beispiel für ein Histogramm mit zweidimensionaler Transferfunktion. Jeder Punkt auf der Transferfunktion hat dort eine Farbe, die dann den Grauwerten zugewiesen wird.

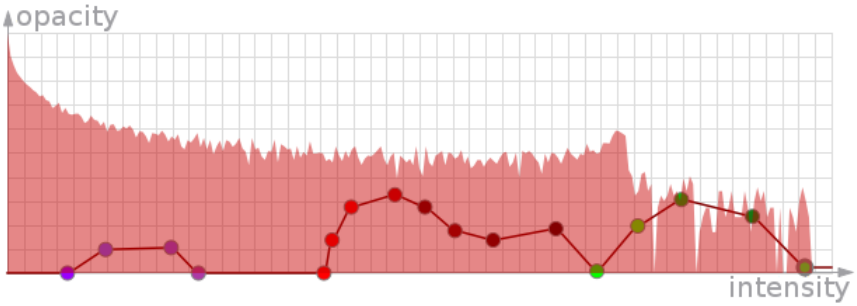


Abbildung 3: Grauwerthistogramm mit zweidimensionaler Transferfunktion
Quelle: eigene Darstellung (erstellt mit VoreenVE)

Abbildung 4 zeigt eine Mischung aus ein- und zweidimensionaler Transferfunktion, bei der es nur möglich ist, die vertikalen Linien zu verschieben, die Höhe zu verändern und die Zuordnungsfarbe zu wechseln. Die Höhe repräsentiert wieder die Opazität.

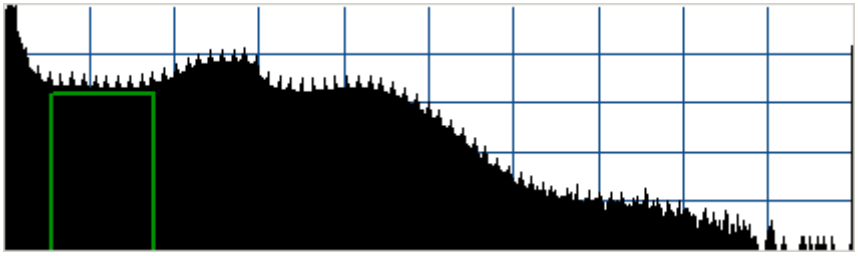


Abbildung 4: Grauwerthistogramm mit vereinfachter zweidimensionaler Transferfunktion

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Neben den Grauwerthistogramm-Transferfunktionen gibt es noch weitere, wie z.B. die gradientenbasierten Transferfunktionen, die für diese Arbeit aber nicht von Bedeutung sind [vgl. PB07, S. 137ff.].

2.2 OpenGL und GLSL

OpenGL (kurz für: **Open G**raphics **L**ibrary) bezeichnet „eine Software Schnittstelle für die grafische Hardware“ [SA10]. Diese Schnittstelle zeichnet sich durch ihre Plattform- und Hardware-Unabhängigkeit aus, da sie von den marktführenden Grafikkartenherstellern als Standard implementiert wird.

Für den Programmierer bedeutet dies, dass er in seiner Anwendung OpenGL Befehle aufruft, die dann vom Grafikkartentreiber interpretiert werden, ähnlich wie in einem Client-Server-Modell[7].

Die Grafikbibliothek ist dabei eher prozedural als deskriptiv, das heißt, anstatt eine Szene und zu zeichnende Objekte zu beschreiben, ruft man wie bei einer Zustandsmaschine[8] nacheinander die einzelnen Schritte auf, die nötig sind, um ein bestimmtes Aussehen oder einen bestimmten Effekt zu erzielen.

OpenGL dient dazu, um zwei- und dreidimensionale Grafiken auf dem Bildschirm darzu-

[7] Das Programm bildet hierbei den Client und der Grafikkartentreiber den Server.

[8] Eine Zustandsmaschine (engl. *state-machine*) behält einen Zustand solange bei, bis er explizit neu gesetzt wird. Analog dazu gilt dies in OpenGL z.B. für Beleuchtungsmodi, Transformationsmatrizen, Farben, etc.

stellen. Dazu werden die Eingabedaten unter Berücksichtigung der aktuellen Zustände (s.o.) verarbeitet und enden im sog. Framebuffer im Speicher der Grafikkarte. Bei den Eingabedaten handelt es sich um eine Menge von sog. Primitiven, mit denen selbst komplexeste 3D Modelle beschrieben werden: Punkte, Linien, Dreiecke, Vierecke und Polygone[9]. Abbildung 5 zeigt eine grobe Skizze dieses Ablaufs, der *Rendering Pipeline* genannt wird. Der gesamte Prozess durchläuft dabei der Reihe nach mehrere Einzelschritte, deren Ergebnisse auch in einem weiteren Schritt zusammenlaufen können.

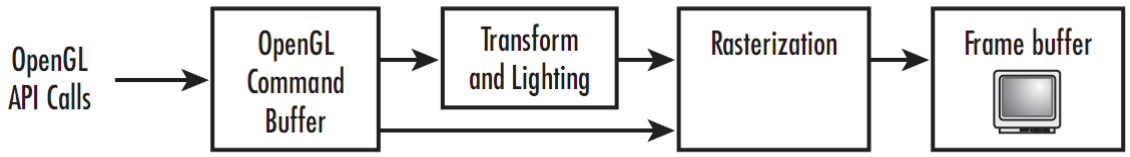


Abbildung 5: Vereinfachte Darstellung der OpenGL Rendering Pipeline
Quelle: [WJLN07, S. 40]

Alle OpenGL Funktionsaufrufe und die Eingabedaten landen zunächst in einer Warteschlange (*OpenGL Command Buffer*), wo sie dann entweder automatisch oder manuell durch den Programmierer an den nächsten Schritt der Pipeline weitergereicht werden. Danach werden die Eingabedaten transformiert und ihre Beleuchtung berechnet (s. Kapitel 2.2.1). Aus diesen Daten wird durch Rasterung dann das tatsächliche Bild erstellt und in den Framebuffer kopiert.
Das Definieren einer Bildschirmregion für die OpenGL Ausgabe ist betriebssystemabhängig, bedient sich aber immer einer Struktur namens *Rendering Context*, um OpenGL Einstellungen und Zustände für diese OpenGL Ausgabe zu speichern [vgl. WJLN07, S. 27, 38ff., 654].

[9] In der 3D Grafik spricht man von einem Vertex (Pl.: Vertices), um die Position eines Punktes (z.B. der Eckpunkt eines Polygons) zu beschreiben.

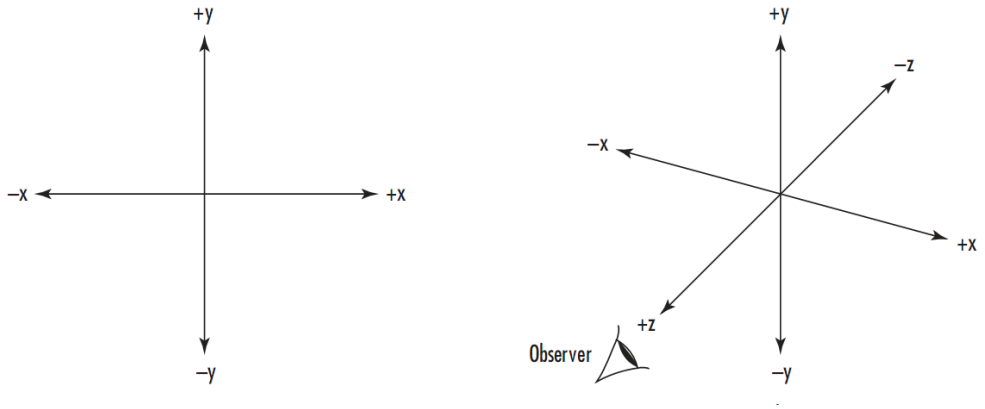
2.2.1 Transformationen

Um dreidimensionale Objekte auf ein zweidimensionales Bild zu projizieren sind einige Koordinatentransformationen[10] nötig:

Viewing Transformation:

Zunächst wird eine Transformation benötigt, um festzulegen, wie die zu rendernde Szene dargestellt werden soll. Dazu wird eine Kameraposition relativ zum Ursprung des Koordinatensystems angegeben, sowie die Blickrichtung und die Rotation der Kamera. Die Blickrichtung dient hierbei als Rotationsachse. Das Ausgangskoordinatensystem[11] der Szene wird damit in den Kameraraum (*Eye Space*) transformiert. Dieses Kamerakoordinatensystem stellt den Blickwinkel des Betrachters dar, also das Ausgangskoordinatensystem, durch die Augen der Kamera gesehen. OpenGL nennt dies die *Viewing Transformation*.

Abbildung 6 zeigt den Koordinatenraum vor und nach der Viewing Transformation.



(a) Koordinatensystem vor der Transformation

(b) Koordinatensystem nach der Transformation

Abbildung 6: View Transformation

Quelle: [WJLN07, S. 129]

[10] Da diese in der Fachliteratur fast ausschließlich auf englisch auftauchen, werden hier ebenfalls die englischen Begriffe der einzelnen Transformationen verwendet.

[11] Das absolute Koordinatensystem, das vor jeglicher Transformation zu Grunde liegt; dabei handelt es sich um ein normales rechtshändiges, dreidimensionales Koordinatensystem: Die x-Achse zeigt nach rechts, die y-Achse nach oben und die z-Achse geradeaus.

Modeling Transformation:

Ein Objekt wird durch seine Vertices definiert, die zunächst in einem eigenen Koordinatensystem, dem sog. Objektraum (*Object Space*), vorliegen — man spricht von Objektkoordinaten. Im Objektraum steht die Position der Vertices relativ zueinander im Vordergrund.

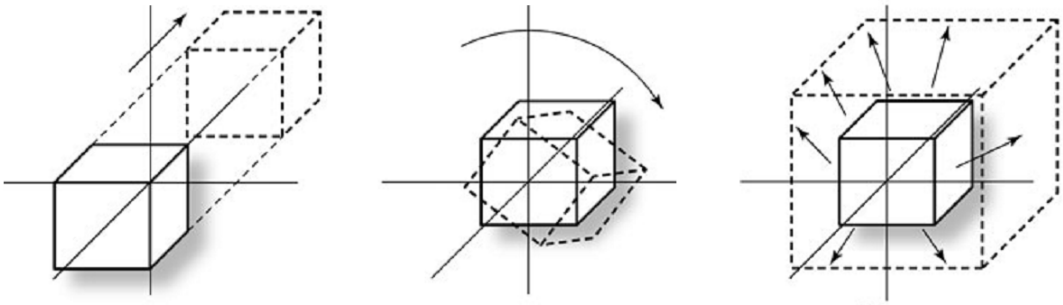
Die Darstellung einer Szene mit mehreren Objekten, die verschiedene Maßstäbe benutzen oder andersartig orientiert sind, erfordert eine Transformation der einzelnen Objektkoordinaten in einen neuen Koordinatenraum derart, dass die Position der Objekte zueinander, ihre relative Skalierung und jeweilige Rotation bestimmt werden. Das bedeutet, sie werden alle in einer Szene platziert, rotiert und skaliert. Man spricht hierbei vom Weltkoordinatensystem (*World Space*) bzw. der *Modeling Transformation*.

Es ist möglich, Informationen über Translation, Rotation und Skalierung gemeinsam in einer 4x4 Matrix darzustellen. Deshalb wird die Transformation von Objekt- in Weltkoordinaten durch die sog. Model Matrix repräsentiert. Um ein Vertex zu transformieren, multipliziert man es (als Vektor) mit der Transformationsmatrix wie folgt:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} x_S \cdot (x_R^2(1-c)+c) & x_R y_R(1-c) - z_R s & x_R z_R(1-c) + y_R s & x_T \\ y_R x_R(1-c) + z_R s & y_S \cdot (y_R^2(1-c)+c) & y_R z_R(1-c) - x_R s & y_T \\ z_R x_R(1-c) - y_R s & z_R y_R(1-c) + x_R s & z_S \cdot (z_R^2(1-c)+c) & z_T \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

Dabei ist $[x, y, z]$ der zu transformierende Vektor, $[x_S, y_S, z_S]$ ist der Skalierungsvektor, $[x_T, y_T, z_T]$ ist der Translationsvektor, $[x_R, y_R, z_R]$ ist der Einheitsvektor, um den rotiert wird, $c = cos(\alpha)$, $s = sin(\alpha)$ und α ist der Rotationswinkel.

Abbildung 7 zeigt jeweils ein Beispiel für jede Modelltransformation.



(a) Translation (b) Rotation (c) Skalierung

Abbildung 7: Modelltransformationen
Quelle: [WJLN07, S. 131]

Modelview Transformation:
Da es für den Betrachter keinen Unterschied macht, ob man ein Objekt z.B. vergrößert oder die Kamera stattdessen näher an das Objekt heranschiebt, werden die *Modeling* und *Viewing Transformation* in der *Modelview Matrix* bzw. *Modelview Transformation* zusammengefasst. Dieser Effekt wird als *Modelview Duality* bezeichnet.

Anschließend folgen zwar noch weitere Transformationen, diese sind aber für diese Arbeit nicht von Bedeutung [vgl. RLK09, S. 23f.].

2.2.2 Texturieren

Unter dem Vorgang des Texturierens versteht man die Zuweisung von Bilddaten zu einem Primitiven[12]. Handelt es sich bei dem Primitiven z.B. um einen Würfel, wird die sog. *Textur* – das zuzuweisende Bild – auf die Oberflächen des Würfels gelegt, wie in Abbildung 8 zu sehen.

[12]Primitive sind geometrische Grundobjekte (z.B. Dreiecke oder Würfel), aus denen sich kompliziertere Formen zusammensetzen lassen.

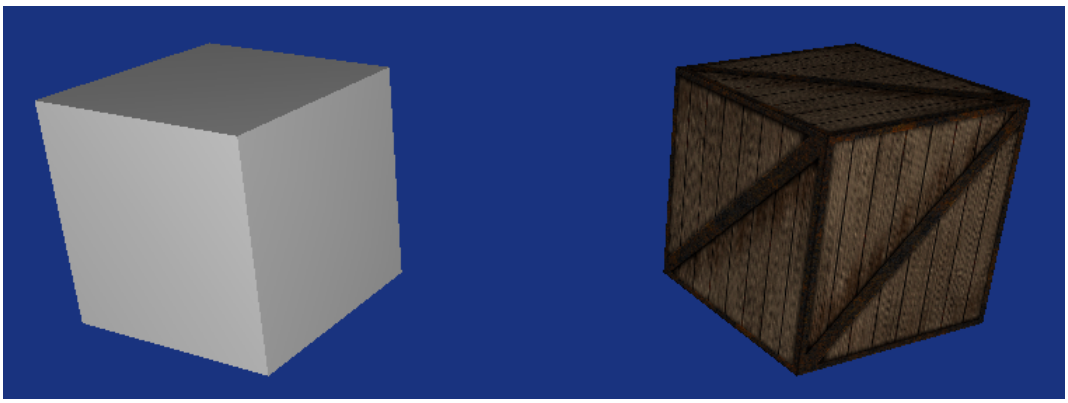


Abbildung 8: Zwei Würfel in einer Szene – der Rechte mit Textur, der Linke ohne
Quelle: eigene Darstellung (erstellt mit Hilfe von OpenTK[13])

Texturen können auch eindimensional (Linien) oder dreidimensional (Volumen) sein, wie in Abbildung 9 dargestellt wird.
Die einzelnen Bildelemente einer Textur werden Texel genannt — in Analogie zu Pixeln[14] bei Bildern. Beim Texturieren (engl. *Texture Mapping*) wird jedem Vertex des Primitiven eine Texturkoordinate zugewiesen. Diese entsprechen den x-, y- und z-Koordinaten von Vertices, werden aber mit s, t und r bezeichnet und durch einen Fließkommawert zwischen 0.0 und 1.0 beschrieben. Abbildung 9 zeigt, wie mit Texturkoordinaten auf Texel zugegriffen wird.

[13] http://www.opentk.com
[14] Der Begriff Pixel leitet sich vom englischen *Picture Element* ab. Texel bedeutet analog dazu *Texture Element* [vgl. WJLN07, S. 303f.].

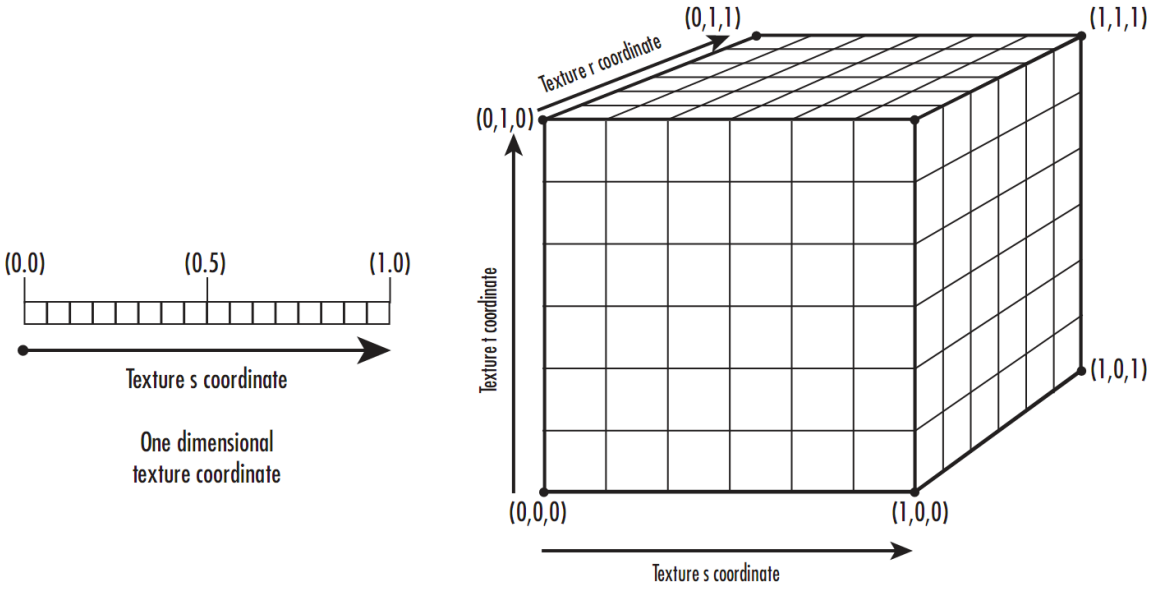


Abbildung 9: Ein- und dreidimensionaler Zugriff auf Texel über Texturkoordinaten
Quelle: [WJLN07, S. 309]

Texturen kommt zusätzlich eine wichtige Funktion als Speicherstruktur hinzu. Wie der spätere Verlauf der Arbeit zeigt, werden sie verwendet, um Bilder oder Volumen im Grafikkartenspeicher zu halten, was besonders im Rahmen der dynamischen Grafikkartenprogrammierung von Bedeutung ist (siehe Kapitel 2.2.3).
In der OpenGL Pipeline (siehe Abbildung 5) ist das Texturieren nach der Rasterung angesiedelt[15] [vgl. WJLN07, S. 303ff.].

2.2.3 OpenGL Shading Language

Ab Version 2.0 wurde der OpenGL Standard um die Open**GL** **S**hading **L**anguage (oder auch: **GLSL**) erweitert. Dabei handelt es sich um eine höhere Programmiersprache, die es dem Programmierer ermöglicht, das Standardverhalten der Grafikkarte teilweise zu verändern und dynamisch nach eigenen Wünschen zu gestalten.

[15] Abbildung 5 zeigt eine stark vereinfachte Darstellung der OpenGL Pipeline, in der das Texturieren nicht eingetragen ist. Anhang A zeigt eine detailliertere Version, in der auch das Texturieren vorkommt.

Zwei Bereiche, in denen dies möglich ist, sind die Verarbeitung von Vertices und von Fragmenten[16]. Gemeint sind damit die Operationen, die auf jedes Vertex bzw. jedes Fragment angewandt werden. Bei Vertices handelt es sich dabei vornehmlich um Transformationen und Beleuchtungskalkulation, bei Fragmenten vor allem um das Texturieren.
Die Bereiche in der OpenGL Pipeline, die GLSL Quellcode interpretieren und bestehende Funktionalität dadurch ersetzen können, nennt man jeweils *Vertex-* und *Fragmentprozessor*. In Abbildung 10 sind die beiden Prozessoren und ihre Position in der OpenGL Pipeline dargestellt.

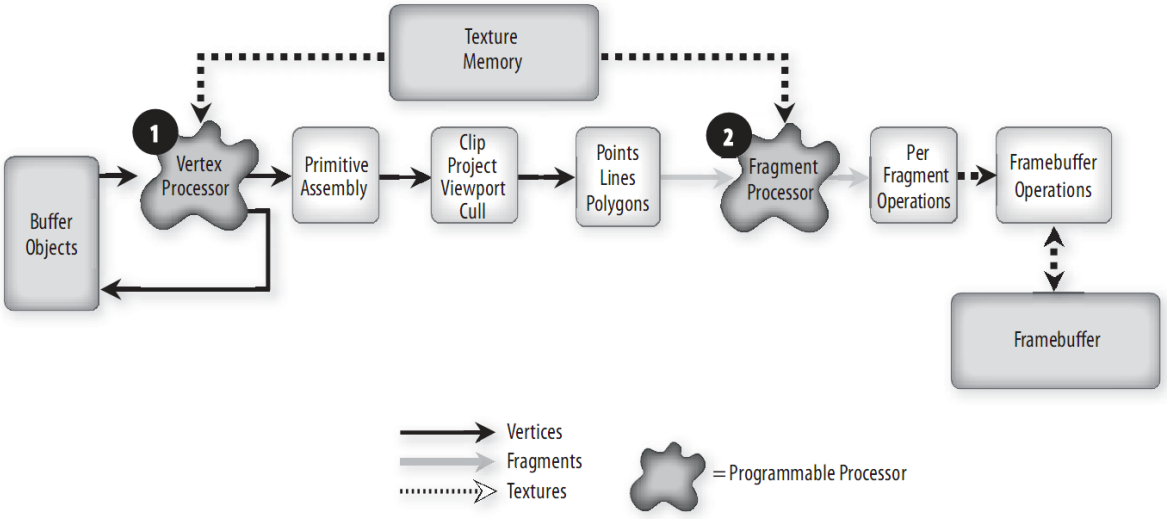


Abbildung 10: OpenGL Pipeline mit Vertex- und Fragmentprozessor
Quelle: [RLK09, S. 39]

Quellcode, der in der OpenGL Shading Language verfasst ist und sich auf dem Vertex- oder Fragmentprozessor ausführen lässt, wird *Shader* genannt. Dementsprechend gibt es für jeden Prozessor eine eigene Art von Shader: *Vertex Shader* und *Fragment Shader* (oftmals auch einfach *Pixel Shader* genannt). Mit Hilfe von OpenGL lassen sich die Shader zu ausführbaren *Programmen* kompilieren, welche dann auf den Prozessoren eingesetzt werden können [vgl. RLK09, S. 35f.].
Mittlerweile gibt es nach der neusten OpenGL Shading Language Spezifikation zwar

[16] Fragmente sind Datenstrukturen auf Pixelebene, die durch die Rasterung von Primitiven entstehen [vgl. RLK09, S. 35].

bereits weitere Prozessoren und Shader [vgl. KBR11, S. 13], jedoch sind diese hier nicht von Relevanz.

Shader Syntax und Beispiel

Shader Code basiert auf der Programmiersprache C und gleicht der Syntax auch in ihren Grundzügen (z.B. ist auch Shader Code prozedural). Jeder Shader besitzt eine *main()* Funktion ohne Rückgabewert als Einstiegspunkt. Im Folgenden wird nur auf Strukturen und Funktionen eingegangen, die für den Fragment Shader Code von Bedeutung und für das Verständnis dieser Arbeit erforderlich sind.
Zusätzlich zu den aus C bekannten Standardtypen (**int**, **float**, **bool**) gibt es noch weitere vordefinierte Typen [vgl. RLK09, S. 69ff.]:

- **Vektoren**: **vec2**, **vec3** und **vec4** sind Vektoren mit jeweils 2, 3 und 4 float Werten als Komponenten. Auf die einzelnen Komponenten kann man wie auf ein Array zugreifen ([] Operator) oder mit den Feldselektoren **.x**, **.y**, **.z**, **.w** oder **.r**, **.g**, **.b**, **.a**. Es gibt außerdem die Möglichkeit, mehrere Komponenten als Vektor zurückzugeben, z.B. mit **.rgb** oder **.xy**.

- **Matrizen**: **mat2**, **mat3** und **mat4** sind Matrizen, die jeweils aus 2x2, 3x3 und 4x4 float Werten bestehen. Sie werden semantisch auch als Matrizen behandelt (z.B. bei der Multiplikation mit Vektoren). Auch auf sie kann mit dem Array Operator zugegriffen werden, egal ob man damit auf einen einzelnen Wert zugreift (z.B. *matrix[1][2]*) oder auf eine ganze Reihe/Spalte (z.B. *matrix[1]*).

- **Sampler**: Ein Sampler kann als Handle[17] für eine Textur verstanden werden. Von Bedeutung sind vor allem **sampler1D**, **sampler2D** und **sampler3D**, die jeweils eine ein-, zwei-, bzw. dreidimensionale Textur repräsentieren.

[17] Gemeint ist der Programmierbegriff Handle (z.B. aus C++), der fast ausschließlich in seiner englischen Form verwendet wird und deshalb auch hier nicht übersetzt wird.

Des Weiteren gibt es Bezeichner, die die Ein-/Ausgabevariablen des Shaders genauer eingrenzen [vgl. RLK09, S. 79]:

- **in**: Eingabevariable, die Werte aus dem Vertex Shader enthält
- **out**: Ausgabevariable
- **const**: Schreibgeschützte Konstante
- **uniform**: Variable, die ihren Wert von der Anwendung erhält; dieser Wert ändert sich nicht sehr oft

Von besonderer Bedeutung sind die speziellen Ein- und Ausgabevariablen eines Fragment Shaders [vgl. KBR11, S. 103ff.]:

- **gl_TexCoord[n]**: Eingabevariable, die die n-ten Texturkoordinaten des Vertex Shaders enthält
- **gl_FragCoord**: Eingabevariable, die die Fensterkoordinaten enthält
- **gl_FragColor**: Ausgabevariable, die die Farbe des Fragments festlegt
- **gl_FragData[n]**: Ausgabevariable, die die Farbe des Fragments für das Renderziel n festlegt[18]

Eine weitere wichtige Funktion ist **texture**, die eine ein-, zwei- oder dreidimensionale Textur abtastet und den Farbwert an den gewünschten Koordinaten zurückgibt [vgl. KBR11, S. 137].

Folgendes Beispiel lädt drei verschiedene Texturen ein und vermischt diese mit Hilfe eines Fragment Shaders. Die erste Textur ist eine Grauwertmaske. Die zweite Textur soll anhand der Maske mit der dritten Textur vermischt werden. Abbildung 11 zeigt die einzelnen Texturen und das Ergebnis. Listing 1 beinhaltet den OpenGL Shading Language Code des Fragment Shaders.

[18] Standardgemäß dient in OpenGL immer der Framebuffer als Ausgabe für das Rendern. Die Ausgabe lässt sich allerdings auch auf sogenannte *Framebuffer Objects* umleiten, so dass es möglich ist, mehrere Renderziele zu haben und z.B. ganze Szenen als Texturen darzustellen. Weitere Informationen zu dem Thema finden sich im Buch [WJLN07, S. 608ff.].

(a) Grauwertmaske

(b) Grastextur

(c) Wegtextur

(d) Ergebnis des Multitextur Shaders

Abbildung 11: Multitextur Fragment Shader mit Eingabetexturen (obere Reihe) und Ausgabe (untere Reihe)

Quelle: eigene Darstellung (erstellt mit OpenTK)

```
uniform sampler2D mask; // Grauwertmaske
uniform sampler2D grass; // Grastextur
uniform sampler2D path; // Wegtextur

void main()
{
    // Texturkoordinaten aus dem Vertex Shader holen
    vec2 texcoord = vec2(gl_TexCoord[0]);

    // Farbwert der Maske an den Koordinaten auslesen
    vec4 gray = texture(mask, texcoord);

    // Die Grasfarbe wird mit der Maskenintensitaet multipliziert, die
    // Wegfarbe mit ihrem Kehrwert. Zusammen ergeben sie die Ausgabefarbe.
    gl_FragColor = texture(grass, texcoord) * gray.r +
                   texture(path, texcoord) * (1.0 - gray.r);
}
```

Listing 1: Multitextur Fragment Shader Code

In diesem Shader werden zunächst die Texturkoordinaten aus dem Vertex Shader geholt. Danach wird der Farbwert (bzw. Grauwert in diesem Fall) für die Maskentextur an den zuvor erhaltenen Koordinaten ausgelesen. Anschließend werden die Gras- und Wegtextur ebenfalls ausgelesen und mit der Grauwertintensität der Maske (bei der Grastextur) bzw. mit ihrem Kehrwert (bei der Wegtextur) multipliziert und als Ausgabefarbe für dieses Fragment festgelegt. Der Farbkanal für Rot wird dabei als Grauwertintensität verwendet, weil bei einem Grauwertbild alle Farbkanäle denselben Wert besitzen.

2.3 Voreen

Bei Voreen (kurz für: Volume Rendering Engine) handelt es sich um ein Open Source Rapid Prototyping [19] Framework zur Visualisierung von Volumendaten, das von der Arbeitsgruppe Visualisierung und Computergrafik (VisCG) am Institut für Informatik der Westfälischen Wilhelms-Universität Münster entwickelt wird, mit dem Hauptziel, den Rendering-Prozess so weit zu modularisieren, dass die Performanz darunter möglichst wenig leidet [vgl. MS+09a].
Das Besondere an Voreen ist, dass der Rendering Prozess mit Hilfe der dynamischen Grafikkartenprogrammierung von OpenGL (siehe Kapitel 2.2.3) auf der GPU stattfindet. Nur deshalb ist es möglich, Volume Rendering in Echtzeit darzustellen.
Die Verwendung des Industriebegriffs „Rapid Prototyping“ erklärt sich dadurch, dass sich mit Voreen geeignete Ergebnisse mit vergleichsweise geringem Aufwand innerhalb kürzester Zeit und ohne viel Erfahrung erzielen lassen. Dies wird deutlich in der objektorientierten Struktur, die dem Framework zugrunde liegt:
Eigenständige, in sich geschlossene Bausteine, sogenannte Prozessoren werden vom Benutzer miteinander zu einem Datenflussnetzwerk verknüpft (siehe Abbildung 12). Ein zentraler Controller namens *NetworkEvaluator* iteriert daraufhin durch die einzelnen Elemente des Netzwerks, leitet den Datenaustausch zwischen den einzelnen Prozessoren

[19] Rapid Prototyping beschreibt eine iterative Methode zur Softwareentwicklung, die darauf abzielt, möglichst früh (in der Entwicklungsphase) brauchbare Ergebnisse zu erzielen mit Hilfe von Prototypen. Oftmals dient Rapid Prototyping der Entscheidungsfindung, ob ein Projekt weiterentwickelt wird oder nicht [vgl. LS92].

ein und führt sie in der richtigen Reihenfolge aus, so dass eine Ausgabe entsteht.
Die Prozessoren werden dabei an sogenannten Ports miteinander verknüpft, um Daten von einem Prozessor zum anderen zu übergeben. Ein Outport bestimmt die Art der Daten, die an den nächsten Prozessor weitergegeben werden und ein Inport bestimmt die Art der Daten, die von diesem Prozessor akzeptiert werden. Die Kommunikation beschränkt sich hierbei hauptsächlich auf die Weitergabe von Volumendaten (OpenGL 3D Texturen) und Render Ausgaben (OpenGL 2D Texturen).
Jeder Prozessor verrichtet seine eigene Aufgabe: Er lädt, erstellt, manipuliert oder verarbeitet Daten und reicht diese weiter, bis das Ergebnis von einem letzten Prozessor dargestellt wird. Eine Liste von speziellen Eigenschaften in jedem Prozessor ermöglicht es dabei, seine Parameter zu verändern.
Mit dieser Architektur lassen sich sehr schnell Ergebnisse erzielen, die sich auch leicht wieder verändern lassen. Deutlich wird dies in dem Tool Voreen Visual Environment (VoreenVE), das im Umfang des Voreen Frameworks mitgeliefert wird. In dieser Anwendung lassen sich solche Netzwerke mühelos per Drag & Drop innerhalb von wenigen Sekunden zusammenklicken. Abbildung 12 zeigt ein beispielhaftes Prozessornetzwerk samt Ausgabefenster.

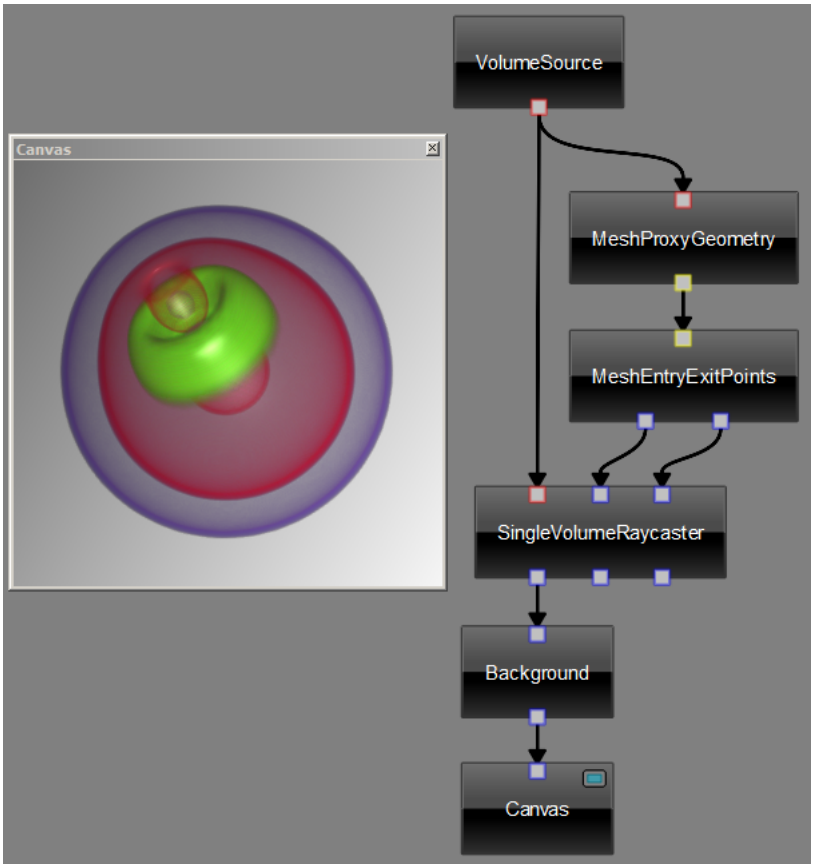


Abbildung 12: Datenflussnetzwerk in VoreenVE mit Ausgabe

Quelle: eigene Darstellung (erstellt mit VoreenVE)

Ein weiterer Vorteil dieses Systems ist die große Flexibilität, denn es lassen sich nach Belieben bestimmte Prozessoren austauschen und neue Prozessoren dazwischen einschieben und untereinander kombinieren, so dass es möglich ist, z.B. mehrere Raycasting Subnetzwerke zu erstellen und miteinander zu verknüpfen damit ein Volumen mehrfach mit verschiedenen Parametern gerendert werden kann (siehe Abbildung 13).

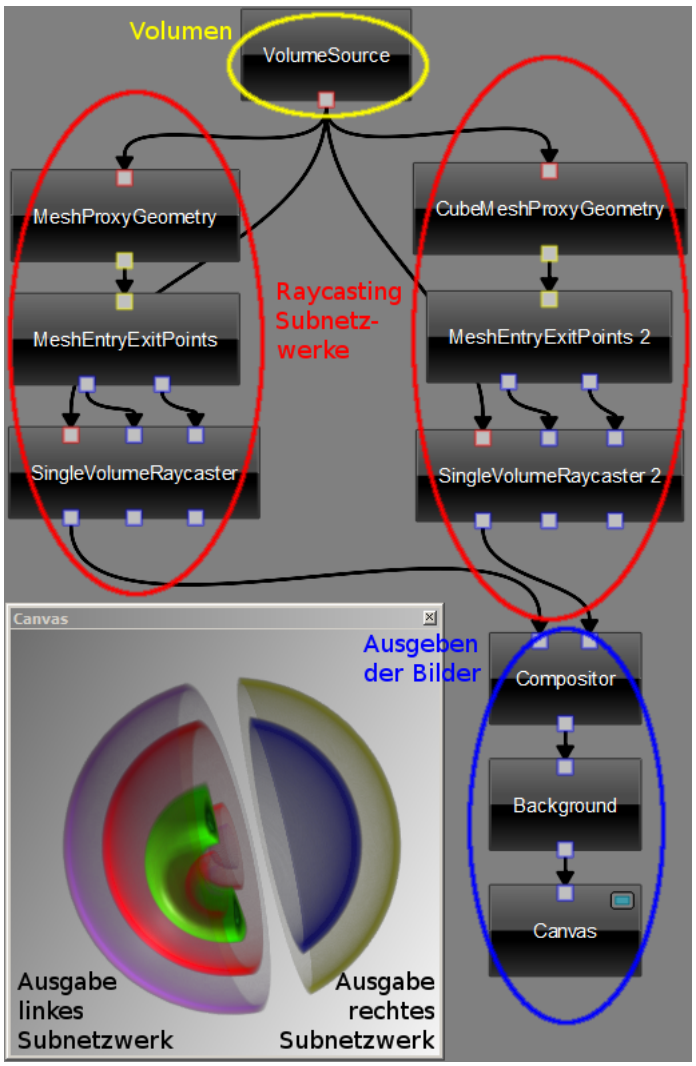


Abbildung 13: Beschriftetes Datenflussnetzwerk mit 2 Raycasting Subnetzwerken mit unterschiedlichen Parametern für Beschneidung und Transferfunktion in VoreenVE mit Ausgabe

Quelle: eigene Darstellung (erstellt mit VoreenVE)

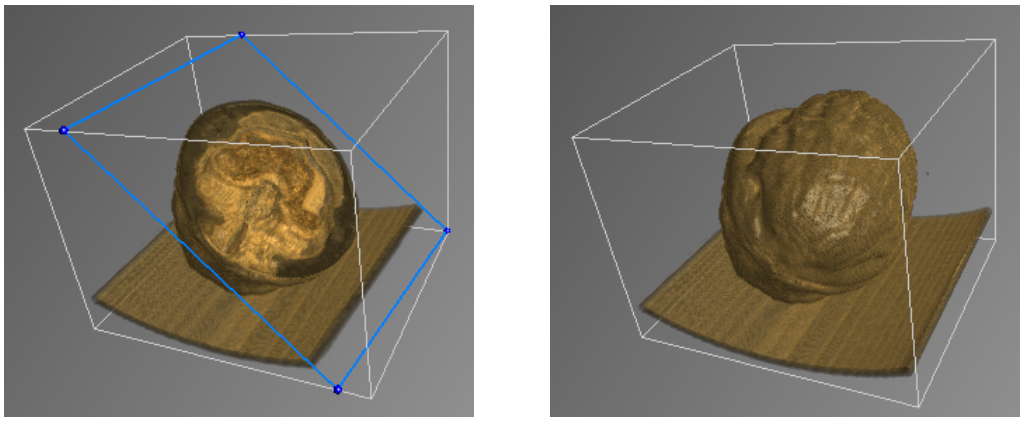
Die wichtigsten Prozessoren in Voreen 2.6.1 sind:

- **VolumeSource**:

  Der VolumeSource Prozessor benötigt keine Eingabe in Form eines Inports. Stattdessen wird ihm über die Eigenschaften eine Volumendatei zugewiesen, die er bei seiner Ausführung einliest und für die Weitergabe an den Outport vorbereitet.

- **CubeMeshProxyGeometry**:

  Dieser Prozessor bekommt ein Volumen über den Inport und umschließt es mit einem Quader, der das Volumen repräsentieren soll. Durch das Verschieben der einzelnen Quaderkanten in den Eigenschaften kann das Volumen dann beliebig beschnitten werden (in der Fachliteratur bekannt als Clipping). Bei den Quaderkanten spricht man auch von Clippingebenen. Abbildung 14 verdeutlicht das Prinzip von Clipping anhand eines Beispiels.



(a) mit Clipping (blaue Clippingebene) (b) ohne Clipping

Abbildung 14: Walnuss mit und ohne Clipping
Quelle: eigene Darstellung (erstellt mit VoreenVE)

  Der Quader wird anschließend an den Outport weitergegeben.

- **MeshEntryExitPoints**:

  Im MeshEntryExitPoints Prozessor werden die farbcodierten Ein- und Austrittstexturen[20] anhand des übergebenen Würfels erstellt und jeweils als 2D Bilddaten an die zwei Outports weitergereicht.

[20] Voreen rendert einen Würfel der Kantenlänge 1 mit den gegenüberliegenden Eckpunkten (0,0,0) und (1,1,1). Der Würfel hat dieselbe Orientierung und Größe wie die 3D Textur (das Volumen) und die Eckpunkte haben ihre eigenen Koordinaten als Farbwert. Von diesem Würfel werden einmal die Vorderseiten und einmal die Rückseiten gerendert und jeweils als Textur gespeichert. Diese sog. Ein-/Austrittstexturen erleichtern und beschleunigen das Erstellen des Suchstrahls. Abbildung 62 in Anhang B zeigt diese Texturen.

- **SingleVolumeRaycaster**:

  Hierbei handelt es sich um den eigentlichen Raycaster[21] Prozessor, das Herzstück der Volumendarstellung. Als Eingabe benötigt er das Volumen, sowie die vorher berechneten Ein- und Austrittstexturen. Mit Hilfe der Transferfunktion (siehe Kapitel 2.1.3) rendert der Raycaster bis zu drei Ausgabebilder, von denen das erste zusätzliche Tiefeninformationen enthält. Diese drei Ausgaben werden entsprechend an die drei Outports weitergeleitet.

- **CoarsenessRenderer**:

  Der CoarsenessRenderer ist ein Prozessor, der die Bildausgabe des Raycasters verschlechtert, um die Performanz zu erhöhen. Während der Interaktion (d.h. bei Translation, Rotation oder Skalierung) spielt die Genauigkeit der Darstellung in der Regel keine große Rolle und kann zugunsten eines Geschwindigkeitszuwachses verringert werden. Der Prozessor sorgt mit der sog. Interaction Coarseness dafür, dass die Ausgabe während der Interaktion grobkörniger ausfällt, dafür aber flüssiger vonstattengeht. Die Intensität der Vergröberung kann als Parameter verändert werden.

- **Combine**:

  Dieser Prozessor dient dazu, mehrere Eingangsbilder zu einem Ausgangsbild zusammenzufassen. Dies wird mit Hilfe einer sog. Blendingfunktion[22] erreicht. In MOPS 3D kommt dieser Prozessor immer dann zum Einsatz wenn es mehr als einen Raycaster (d.h. mehrere Bilder, die dargestellt werden müssen) gibt.

- **Canvas**:

  Der Canvas (zu deutsch: Leinwand) Prozessor bildet immer den abschließenden Teil des Datenflussnetzwerks und dient nur zur Ausgabe der gerenderten Daten. Er besitzt lediglich einen Inport (keinen Outport), über den zweidimensionale Bild-

[21] Im Raycaster werden die Suchstrahlen durch das Volumen gesandt (siehe Kapitel 2.1.1).

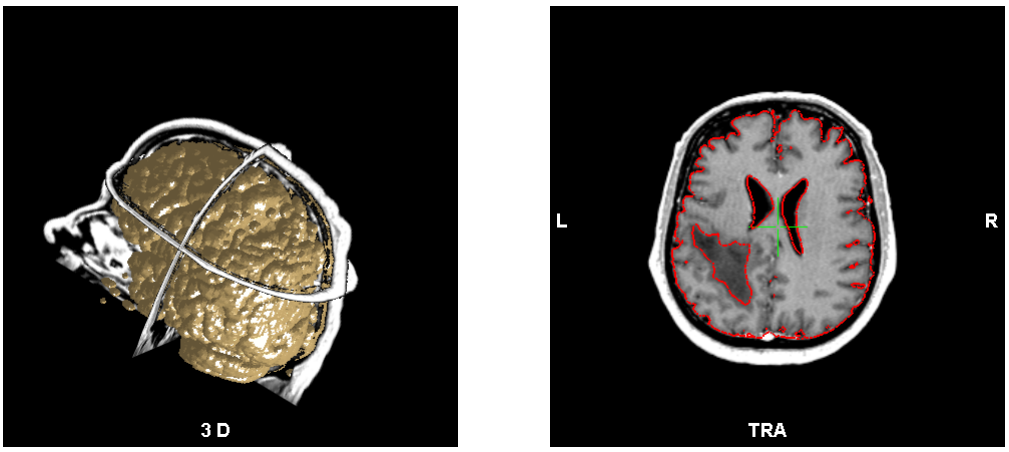
[22] Unter Blending versteht man das Vermischen mehrerer (üblicherweise zweier) Farbwerte an derselben Pixelposition, so dass ein neuer Farbwert entsteht. Es gibt verschiedene Blendingfunktionen, wie z.B. *Alpha Blending*, bei der die Transparenz des Farbwertes seinen Beitrag zum Ergebnis bestimmt, oder *Additives Blending*, bei der die Farbwerte einfach addiert werden.

daten eingehen, die von ihm dann auf dem aktiven OpenGL Rendering Context ausgegeben werden.

2.4 MOPS 3D

Das **M**ultimodale **O**perations**p**lanungs**s**ystem **MOPS** 3D wurde am Institut für medizinische Biometrie und Informatik der Universtität Heidelberg entwickelt. Durch enge Zusammenarbeit mit der neurochirurgischen Abteilung des Universitätsklinikums Heidelberg steht die klinische Anwendung im Vordergrund und das Programm genießt eine hohe Akzeptanz unter Chirurgen.
Das Ziel von MOPS 3D ist es, den Neurochirurgen bei der Planung von schwierigen Operationen, wie z.B. der Tumorentfernung, zu unterstützen und es ihm zu ermöglichen, sich ein Bild der räumlichen Strukturen zu machen. Die Neurochirurgie erfordert besondere Sorgfalt in der Planung, da es bei den Eingriffen sonst leicht zu Verletzungen wichtiger, benachbarter Hirnstrukturen kommen kann. Weiterhin kann MOPS 3D auch intraoperativ eingesetzt werden, z.B. bei der Navigation.
MOPS 3D ist in der Lage, medizinische Bilddatensätze (z.B. MRT, fMRT[23], MEG[24]) zu visualisieren und stellt dem Anwender viele Hilfsmittel zur Verfügung, wie z.B. Segmentierung und Registrierung. Abbildung 15 zeigt einen beispielhaften Einblick in die Visualisierung in MOPS 3D.

[23] funktionelle Magnetresonanztomographie; bei der fMRT wird zusätzlich neuronale Aktivität in Form von Durchblutungsänderungen dargestellt.
[24] Magnetoenzephalographie: Darstellung der magnetischen Hirnaktivität.



(a) Dreidimensional (b) Transversal

Abbildung 15: Darstellung des segmentierten Cortex
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Die Anwendung ist in C++ geschrieben und basiert auf den Microsoft Foundation Classes[25], weshalb sie in Microsoft Visual Studio exklusiv für Windows entwickelt wird. Die zwei- und dreidimensionalen Visualisierungen der MR-Schichten werden mit OpenGL (siehe Kapitel 2.2) und VGL[26] realisiert. Letzteres ist zusätzlich für das Management der Volumen[27] verantwortlich. Die einzige Ausnahme bei der 3D Visualisierung bildet die Volume Rendering Komponente von MOPS 3D — diese setzt auf das integrierte Open Source Framework Voreen (siehe Kapitel 2.3) [vgl. Met+06].

2.4.1 Funktionelle Übersicht

Bilddaten sind in MOPS 3D eindeutig einer Operationsplanung zugeordnet, die wiederum einem Patienten zugeordnet ist. Die Anzeige der Bilddaten erfolgt entweder im Einzel- oder Mehrfachansichtsfenster. Im Letzteren werden dabei vier verschiedene Ansichten dargestellt (siehe Abbildung 16). Eine Ansicht stellt einen Blickwinkel auf die Bilddaten dar und kann folglich transversal, sagittal, coronal, dreidimensional oder chir-

[25] Eine Klassenbibliothek zur grafischen Oberflächenentwicklung für Windows.
[26] Volume Graphics Library: Eine Grafikbibliothek für Volumen- und Oberflächendarstellung.
[27] VGL bietet z.B. eigene Datenstrukturen, um Volumendaten im Hauptspeicher zu halten.

urgisch[28] sein.

In den zweidimensionalen Ansichten (transversal, sagittal, coronal) wird jeweils die entsprechende Schicht der Bilddaten angezeigt. Es wird dabei immer nur jeweils eine Schicht dargestellt – die *aktuelle* Schicht – , jedoch lassen sich alle Schichten einer Ansicht mit dem Mausrad durchblättern.

In der dreidimensionalen Ansicht werden die drei aktuellen Schichten relativ zueinander dargestellt. In allen Ansichten ist es möglich, die dargestellte Szene mit der Maus zu verschieben und skalieren. Einzig allein die Rotation ist der 3D Ansicht vorbehalten. Weiterhin befindet sich in jeder Ansicht ein grünes Fadenkreuz, das immer genau dort sitzt, wo sich die Schichten aller drei 2D Ansichten kreuzen. Es lässt sich in jeder Ansicht beliebig verschieben, so dass sich folglich auch die aktuellen Schichten verändern. Abbildung 16 zeigt die Mehrfachansicht mit dem versetzten Fadenkreuz.

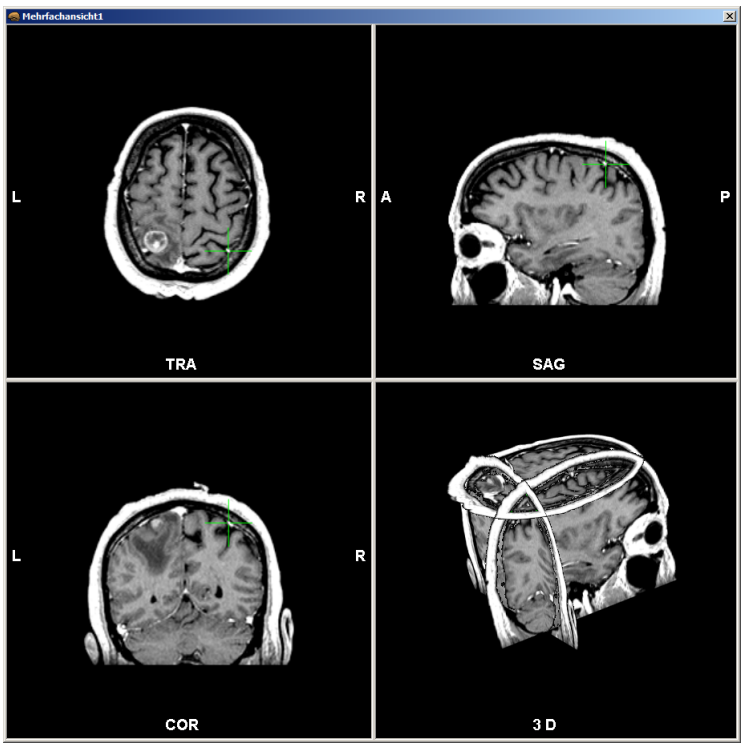
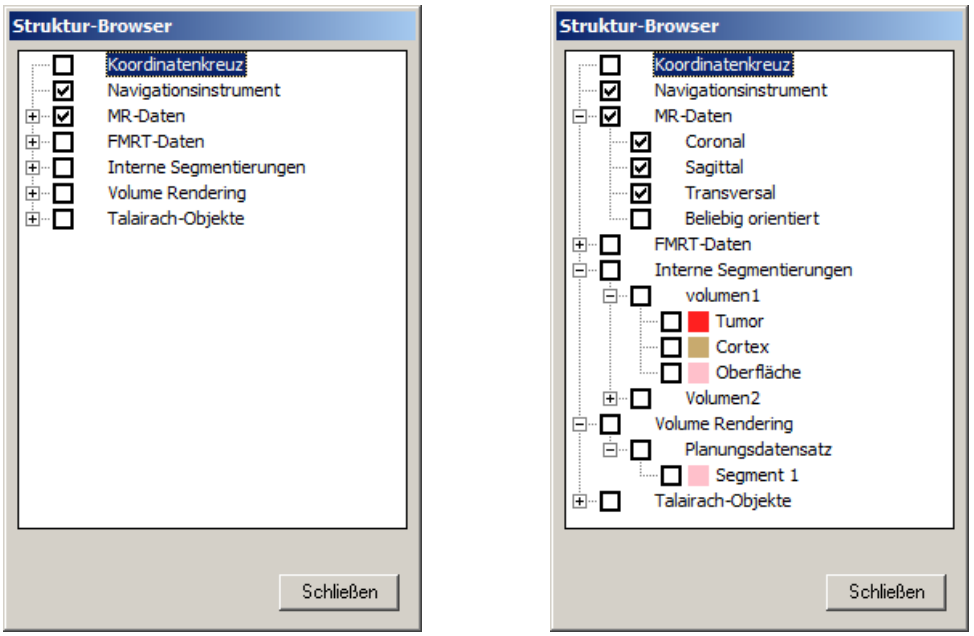


Abbildung 16: Mehrfachansicht: transversal (oben links), sagittal (oben rechts), coronal (unten links) und dreidimensional (unten rechts)

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

[28] Aus Sicht des Chirurgen während der intraoperativen Navigation.

Ein wichtiger Bestandteil von MOPS 3D ist der Strukturbrowser, mit dem sich einzelne Subkomponenten des Volumens, wie z.B. Segmentierungen, MR-Daten oder Volume Rendering Segmente, (de-)aktivieren und konfigurieren lassen. Er besteht aus mehreren Hauptkategorien, die sich teilweise erweitern lassen, um ihre jeweils untergeordneten Elemente anzuzeigen. Dabei wird der Status der einzelnen Elemente (d.h. ihre Konfiguration und ob sie aktiv sind oder nicht) für jede Operationsplanung individuell gesichert. Abbildung 17 zeigt den Strukturbrowser in verschiedenen Zuständen.



(a) Normale Ansicht (b) Ausgeklappt

Abbildung 17: Strukturbrowser in verschiedenen Zuständen

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

An dieser Stelle sei noch jener Dialog erwähnt, der zur Erstellung von Segmentierungen gedacht ist. Für diese Arbeit von Bedeutung ist sein Aufbau und seine Fähigkeit, mit Hilfe von Region Growing[29] und Freihandzeichnen zu segmentieren. In Abbildung 18 ist der Segmentierungsdialog zu sehen.

[29] Region Growing ist ein Segmentierungsalgorithmus, der mit ein oder mehreren Saatpunkten startet und nach bestimmten Kriterien wächst, um Bildelemente (in diesem Fall Voxel) einer Region zuzuweisen [vgl. GWE04, S. 408].

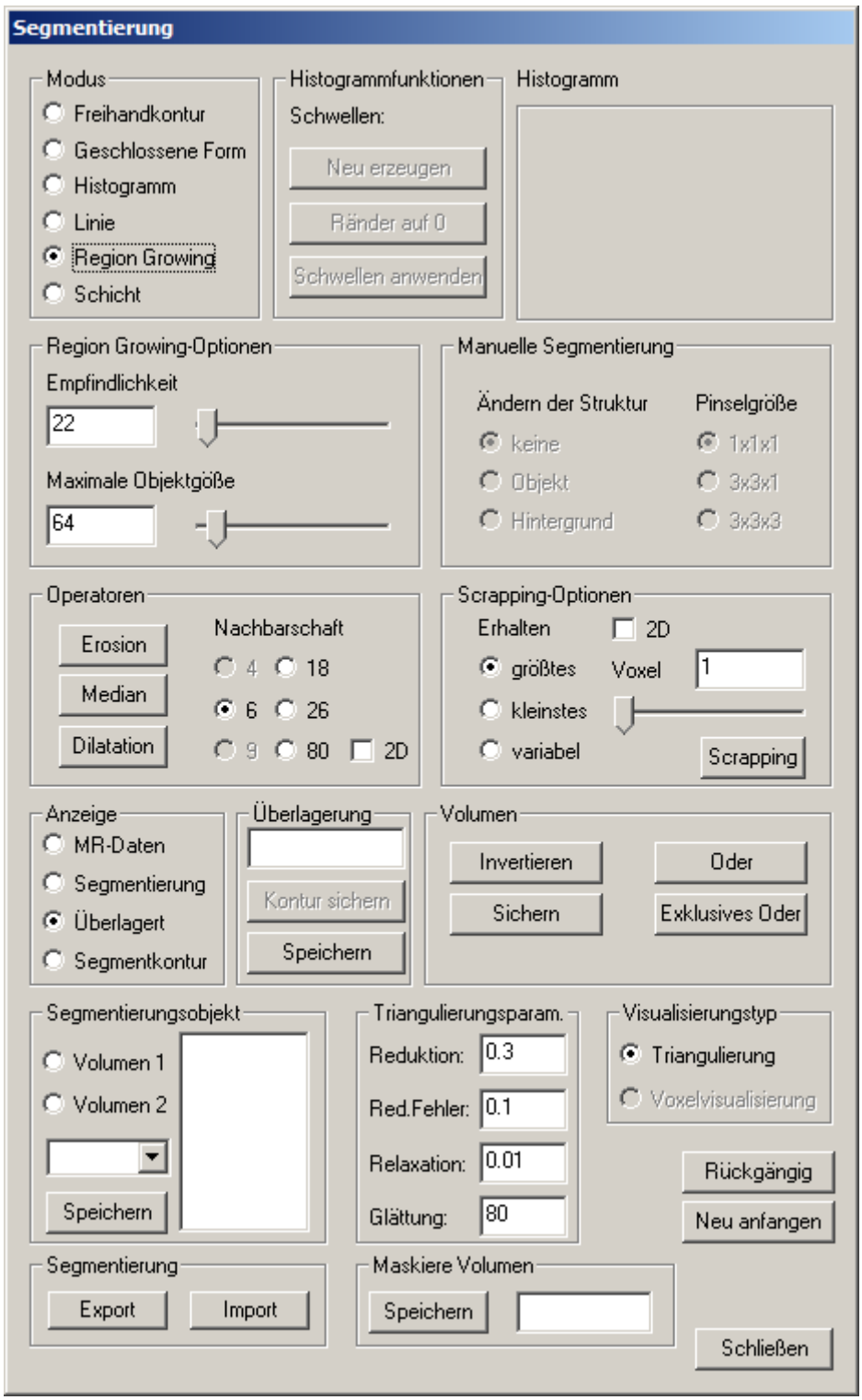


Abbildung 18: Segmentierungsdialog

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Das Segmentieren per Region Growing läuft standardmäßig so ab, dass der Benutzer zunächst „Region Growing“ als Modus auswählt, die gewünschte Empfindlichkeit und max. Objektgröße unter den „Region Growing-Optionen“ einträgt und anschließend in die 2D Ansicht klickt, um einen Saatpunkt (Startpunkt) für das Region Growing zu setzen.

2.4.2 Volume Rendering Komponente

Von besonderer Bedeutung ist die Volume Rendering Komponente von MOPS 3D, die verbessert und erweitert werden soll. Dieses Kapitel gibt einen Überblick über ihre Funktionalität.

Bevor irgendetwas mit Volume Rendering dargestellt werden kann, muss es im Strukturbrowser (siehe Abbildung 17) global aktiviert werden, damit Voreen initialisiert wird. Ein Klick auf das Wort „Volume Rendering“ öffnet den globalen Konfigurationsdialog, in dem sich Darstellungsqualität, Interaktionsqualität und Shading (Beleuchtung, siehe Kapitel 2.1.2) für Volume Rendering regulieren lassen. Shading lässt sich ein- und ausschalten und für die beiden Qualitätseinstellungen gibt es jeweils mehrere vordefinierte Qualitätsstufen (niedrig, mittel, hoch) zwischen denen gewählt werden kann.

Abbildung 19 zeigt den globalen Volume Rendering Konfigurationsdialog.

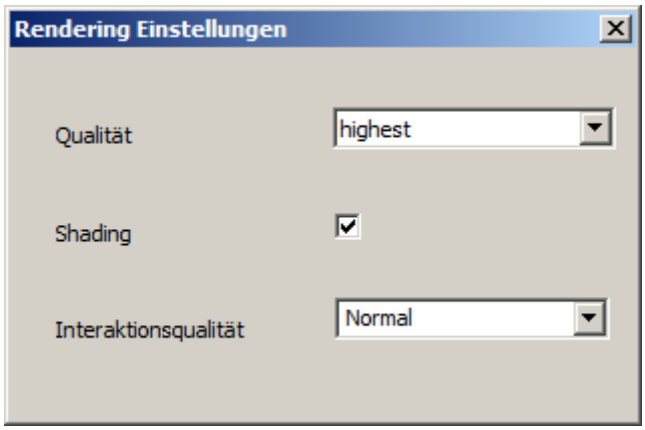


Abbildung 19: Dialog für globale Volume Rendering Einstellungen

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Aufklappen des Volume Rendering Elements im Strukturbrowser enthüllt eine Unterkategorie für jedes geladene Volumen — in der Regel nur eines, nämlich der Planungsdatensatz. Abbildung 20 zeigt die Datenstruktur, die der Hierarchie des Volume Rendering Elements im Strukturbrowser zugrunde liegt.

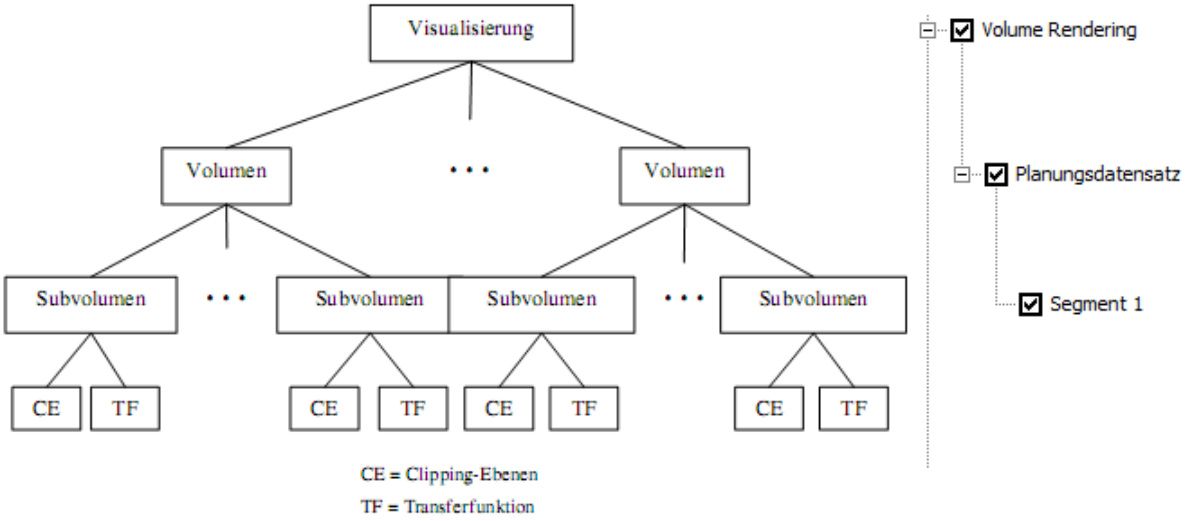


Abbildung 20: Datenstruktur der Volume Rendering Komponente (links) mit entsprechendem Strukturbrowser Beispiel (rechts)

Quelle: [Fre09, S. 55] (links), eigene Darstellung (rechts)

Ein Klick auf den Namen („Planungsdatensatz“) öffnet den Dialog zum Anlegen eines neuen Segments. Jedes Volumen kann beliebig viele Subvolumen (Segmente) besitzen, die sich alle durch ihre Eigenschaften unterscheiden. Beim Anlegen eines neuen Subvolumens wird eine Bezeichnung sowie eine zugeordnete Farbe benötigt. Abbildung 21 zeigt den Dialog zum Hinzufügen eines neuen Subvolumens.

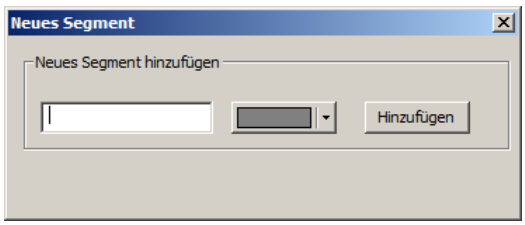


Abbildung 21: Dialog zum Hinzufügen eines neuen Volume Rendering Segments

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Jedes Volumen besitzt initial bereits ein Segment, das sich nicht löschen lässt. Im Strukturbrowser lässt sich jedes Segment über das nebenstehende Häkchen aktivieren und deaktivieren, d.h. darstellen oder ausblenden. Ein Klick auf den Namen eines Segmentes öffnet den Segment-Editor. Dort können die Eigenschaften, die das Segment ausmachen, bearbeitet werden. Abbildung 22 zeigt den Segment-Editor Dialog.

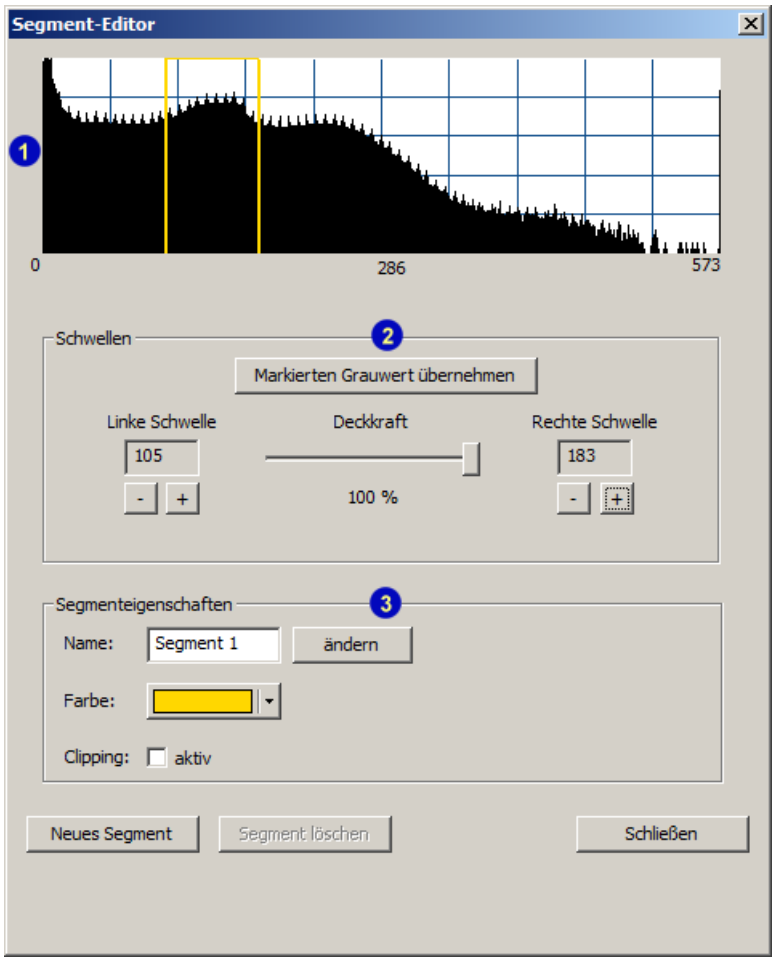


Abbildung 22: Dialog zum Editieren der Eigenschaften eines Volume Rendering Segments

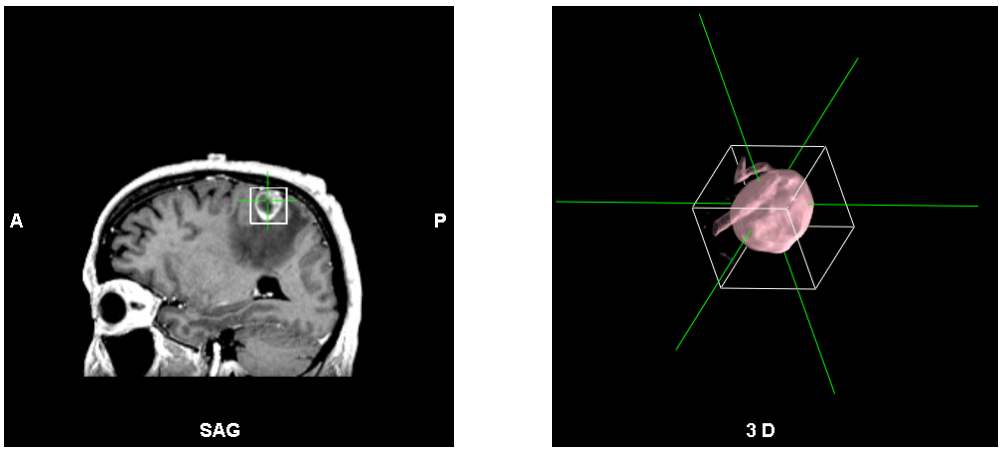
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Der obere Teil des Dialogs (1) zeigt das Grauwerthistogramm. Die Transferfunktion ist in MOPS 3D nicht eine einzelne komplexe Funktion, sondern jedes Segment besitzt seine eigene einfache Transferfunktion. Diese besteht lediglich aus zwei Schwellwerten, so dass

nur jene Grauwerte in das Volume Rendering mit einbezogen werden, die innerhalb der Schwellwerte liegen. Neben der Vereinfachung hat diese Aufteilung den Vorteil, dass Segmente, die bereits gut visualisiert sind, nicht versehentlich durch Bearbeiten der einzigen Transferfunktion verändert werden können [vgl. Fre09, S. 54]. Die Schwellwerte lassen sich über die +/- Buttons in- bzw. dekrementieren oder direkt im Histogramm mit der Maus verschieben.

Der mittlere Teil (2) bietet Möglichkeiten, die Transferfunktion mit mehr Genauigkeit als mit der Maussteuerung zu beeinflussen. Die Opazität lässt sich über einen Schieberegler prozentgenau verstellen und die Schwellwerte über die +/- Buttons (siehe oben). Außerdem gibt es einen Button, der die Schwellen automatisch um den Grauwert herum positioniert, über dem sich das grüne Fadenkreuz gerade befindet. Im unteren Teil (3) lassen sich weitere Eigenschaften, wie Name und Farbe des Segments und Clipping verändern. Clipping wird hier nur ein- oder ausgeschaltet. Die Position und Größe des Clippingquaders wird direkt in den 2D Ansichten gesteuert. Bei aktiviertem Clipping wird der Quader dort mit weißen Linien dargestellt. Mit der Maus lässt er sich entweder komplett verschieben oder nur einzelne Kanten. Bei gedrückter STRG-Taste kann man den Quader mit der Maus neu aufziehen. Der zuvor angezeigte Quader wird dann überschrieben. Der Quader wird auch in der 3D Ansicht mit Linien dargestellt. Abbildung 23 zeigt ein Volume Rendering Segment mit aktiviertem Clipping.

Abschließend gibt es am unteren Ende des Dialog noch die Möglichkeit ein neues Segment zu erstellen oder das alte zu löschen.



(a) Sagittalansicht (b) 3D Ansicht

Abbildung 23: Volume Rendering Clippingquader um einen Tumor (mit 50% Deckkraft)
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Anhang C zeigt den vollständigen Ablauf der Nutzerinteraktion für die Volume Rendering Komponente.

3 Anforderungsanalyse und Konzeption

In diesem Kapitel werden Konzepte ausgearbeitet, um die Volume Rendering Komponente in MOPS 3D funktional zu erweitern, ihre Performanz zu erhöhen und ihre Bedienbarkeit zu vereinfachen, damit sie in der klinischen Anwendung von größerem Nutzen ist. Dazu wird sie analysiert, um mögliche Schwächen aufzuzeigen und entsprechende Verbesserungsvorschläge zu entwickeln.
Um alle Vorteile von Volume Rendering auszunutzen, soll zusätzlich ein GPU-basiertes Segmentierungsverfahren in MOPS 3D integriert und mit der Volume Rendering Komponente verknüpft werden. Dieses Verfahren wird parallel zu dieser Arbeit von Marius Wirths im Rahmen seiner Diplomarbeit entwickelt. Seine Aufgabe besteht darin, ein Framework bereitzustellen, das Segmentierung auf der Grafikkarte ermöglicht, d.h. er implementiert entsprechende Algorithmen hierfür und stellt wichtige Funktionen wie z.B. die Filterung zur Verfügung. In dieser Arbeit dagegen soll das Framework in MOPS 3D integriert werden, so dass die Segmentierungsergebnisse mit Volume Rendering dargestellt werden. Weiterhin soll eine Benutzeroberfläche entworfen werden, die sich am Dialog der bereits vorhandenen Segmentierungsfunktion in MOPS 3D (siehe Abbildung 18) orientiert. Im Gegensatz zu diesem Dialog soll hierbei aber besonders Wert auf Bedienbarkeit und Simplizität gelegt werden. Aufgrund des Umfangs wird diese Aufgabe unabhängig von den anderen Verbesserungen behandelt, obwohl sie prinzipiell auch als funktionale Erweiterung der Volume Rendering Komponente betrachtet werden könnte. Auf die oben genannten Punkte wird in den folgenden Unterkapiteln näher eingegangen.

3.1 Performanz

Die Volume Rendering Komponente wird in diesem Unterkapitel hinsichtlich der Performanz untersucht. Dabei sollen Situationen aufgezeigt werden, in denen sich ihre Performanz verbessern lässt. Zu jeder dieser Situationen wird dann jeweils ein Konzept zur Leistungssteigerung entworfen.
Alle Tests bezüglich der Performanz werden auf einem Computer mit folgenden technischen Daten ausgeführt:

Grafikkarte: nvidia GeForce GTX 260 (896 MB)

Prozessor: AMD Phenom II X4 B50 (3,1 GHz)

Arbeitsspeicher: 4 GB

Betriebssystem: Microsoft Windows 7 64 Bit

Zum Zeitpunkt der Tests ist diese Hardware im Vergleich etwa als mittelklassig einzustufen.

3.1.1 Voreen

Analyse:
Bei dem in MOPS 3D integrierten Voreen Framework handelt es sich um die Version r13 vom 13. Februar 2009. Diese Version weist den großen Nachteil auf, dass sie auf der Testmaschine (bzw. auf allen Rechnern mit nvidia Grafikkarten) nur dann initialisiert und somit erfolgreich ausgeführt werden kann, wenn die Versionsnummer von OpenGL auf dem System kleiner gleich 2.1 ist. Laut Aussage der Entwickler unterstützt Voreen zwar grundlegend auch Grafikkarten von AMD/ATI, doch kommt es dort häufig zu Abstürzen und unvorhersehbarem Verhalten aufgrund von Treiberproblemen (siehe [Vorc]). Der Nachfolger von OpenGL 2.1 (Version 3.0) erschien bereits am 11. August 2008 (siehe [SA10, S. 401]) und ist mittlerweile allgemein verbreitet (die aktuellste Version ist 4.2). Weiterhin kommt es bei der Volume Rendering Komponente in MOPS 3D vereinzelt zu nicht reproduzierbaren Abstürzen und fehlerhaftem Verhalten. Hier liegt die Vermutung

nahe, dass dieses Auftreten Voreen zuzuschreiben ist.

Konzept:
Es erscheint demnach angebracht, die neueste Version von Voreen (2.6.1 vom 4. November 2010) in MOPS 3D zu integrieren. Dies sollte zu einem leichten Leistungsanstieg führen, da Voreen 2.6.1 sich nicht auf OpenGL 2.1 Technologien beschränkt und im Vergleich zu Voreen r13 einige Verbesserungen bezüglich der Performanz beinhaltet (siehe [Vora]).
Ein wichtiger Aspekt bei der Integration soll Flexibilität für zukünftige Upgrades sein. Das heißt, es soll für den Entwickler möglichst wenig Aufwand bedeuten, neue Versionen von Voreen in MOPS 3D zu integrieren.

3.1.2 Deaktivierte Segmente

Analyse:
Vergleicht man in MOPS 3D Volume Rendering mit der herkömmlichen Oberflächendarstellung, lässt sich beim Ersteren ein deutlicher Einbruch in der Performanz feststellen. Rotiert man auf der Testmaschine den gleichen Datensatz wenige Sekunden lang mit der Maus, so kommt die 3D Ausgabe bei eingeschaltetem Volume Rendering auf durchschnittlich 40 Bilder pro Sekunde und ohne Volume Rendering auf 65. Der Test wird mit genau einem Volume Rendering Segment durchgeführt, dessen Transferfunktion alle Grauwerte beinhaltet. Diese Performanz ist absolut vertretbar für ein so ressourcenintensives Verfahren wie Volume Rendering und ist ausreichend für eine flüssige Darstellung auf der mittelmäßig bestückten Testmaschine. Wiederholt man den Test bei aktiviertem Volume Rendering und deaktiviertem Segment, so fällt auf, dass die gleiche Geschwindigkeitseinbuße festzustellen ist – obwohl ein ähnliches Resultat wie bei deaktiviertem Volume Rendering zu erwarten wäre.

Konzept:
Dieses Verhalten ist inakzeptabel für den Nutzer, denn eine Deaktivierung sollte bedeuten, dass das inaktive Objekt nicht unnötig Rechenleistung beansprucht.
Das Problem liegt darin, dass ein deaktiviertes Segment zwar nicht dargestellt, aber dennoch berechnet wird. Die Darstellung wird durch geschickte Wahl der Clippingebenen verhindert, so dass das ganze Segment weggeschnitten wird und somit unsichtbar ist.
Die Deaktivierung soll insofern abgeändert werden, dass das Segment gar nicht erst berechnet wird. Ein geeigneter Ansatz wäre hier, schon früh im Prozessornetzwerk des Segments festzustellen, ob dieses Segment deaktiviert ist und ggf. ein Ausführen der weiteren Prozessoren zu unterbinden.

3.2 Funktionale Erweiterung

In diesem Unterkapitel werden Teilbereiche der Volume Rendering Komponente analysiert, die durch eine Erweiterung ihrer Funktionalität von großem Nutzen für den Anwender wären. Zu jedem dieser Teilbereiche wird daraufhin ein Konzept für den Ausbau seiner Funktionalität erstellt.

3.2.1 Form der Clippingstruktur

Analyse:
Clipping von Volume Rendering Objekten wird eingesetzt, um bestimmte Strukturen (z.B. ein Organ) isoliert darzustellen. Die einzige Konfiguration, die dabei verändert werden kann, ist die Verschiebung der Kanten eines Quaders (siehe Kapitel 2.3 unter *CubeMeshProxyGeometry*). Dies führt zu ausreichenden Ergebnissen bei Strukturen, deren Form ähnlich kantig ist wie ein Quader (z.B. ein Zahn). Häufig müssen allerdings Strukturen segmentiert werden (z.B. oval-rundliche Tumoren), für die die Passform eines Quaders ungeeignet ist, da viel Unerwünschtes nicht weggeschnitten wird. Abbildung 24 veranschaulicht diese Problematik.

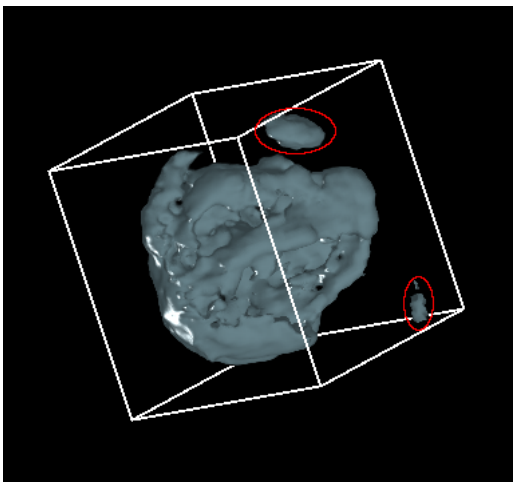


Abbildung 24: Rundlicher Tumor mit Clippingquader; die rot umrandeten Strukturen werden vom Quader miterfasst und deshalb nicht weggeschnitten

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Konzept:

Die Erweiterung des Clipping um weitere geometrische Passformen erscheint hier sinnvoll. Der Benutzer soll im Segment-Editor eine geeignete Geometrie für das Clipping auswählen können, woraufhin sich der beschnittene Anteil des Segments entsprechend verändert. Ein Quader kann dabei weiterhin zur Verschiebung der Grenzen und zum erneuten Aufziehen der Clippingstruktur in den 2D Ansichten dienen.

Das Clipping wird beispielhaft durch ein Ellipsoid erweitert – als Alternative zum Quader –, weil es viele medizinische Strukturen abbilden kann (z.B. Augen, Aneurysmen, Tumoren, u.Ä.) und seine Struktur nicht sehr komplex ist.

Bei der Implementierung des Ellipsoids soll besonders auf die Erweiterbarkeit der Clippingstrukturen geachtet werden, damit es dem Entwickler in Zukunft möglich ist, dem Clipping ohne viel Aufwand weitere geometrische Strukturen hinzuzufügen.

3.2.2 Inverses Clipping

Analyse:

Eine weitere Einschränkung des Clipping liegt darin, dass nur der Teil außerhalb der

Clippingstruktur (Quader) weggeschnitten wird (siehe Abbildung 23). Es wäre aber sinnvoll, auch die Möglichkeit zu haben, alles wegzuschneiden, was sich innerhalb der Clippingstruktur befindet, also das Clipping zu invertieren. So wäre der Nutzer imstande, sich Strukturen von innen heraus zu betrachten, beispielsweise den Cortex.

Konzept:
Wie bereits in der Analyse erwähnt, sollte es möglich sein, beim Clipping eines Segments festzulegen, ob der Bereich innerhalb oder außerhalb der Clippingstruktur weggeschnitten werden soll. In Abbildung 23 zum Beispiel würde das bedeuten, dass alles sichtbar wäre bis auf den Tumor innerhalb des Clippingquaders. Abbildung 25 verdeutlicht dieses Szenario (das Rechteck, in dem sich der Tumor befand, wurde nachträglich mit einem Bildbearbeitungsprogramm herausgeschnitten).

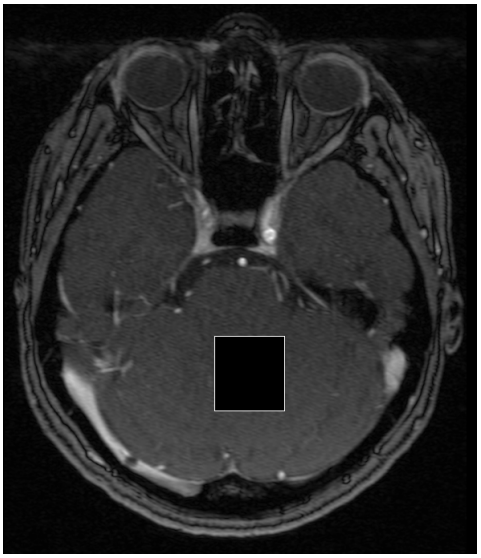


Abbildung 25: Konzeptbild zum inversen Clipping: Alles außer dem Tumor ist sichtbar
Quelle: eigene Darstellung

Auf diese Weise ließen sich auch gezielt Strukturen entfernen, wie z.B. die rot umrandeten Strukturen in Abbildung 24.
Ein Anwendungsgebiet dafür ist beispielsweise die CT-Angiographie, bei der die Gefäße aufgrund eines Kontrastmittels ähnlich starke Grauwertintensitäten aufweisen wie

Knochen. Ziel ist es, die Gefäße isoliert darzustellen. Die Ähnlichkeit der Intensitäten führt im Volume Rendering dazu, dass die Transferfunktion des Segments nicht so eingestellt werden kann, dass sie nur die Gefäße erfasst, sondern auch die Knochen. Weiterhin gibt es keine einfache Clippingstruktur, welche die Struktur von Gefäßen gut abbildet. Aus diesem Grund ist es oftmals schwer möglich, die Gefäße getrennt von den Knochen darzustellen. Hierbei wäre es einfacher, die irrelevanten Strukturen mit mehreren Clippingstrukturen zu erfassen und mit inversem Clipping auszublenden, anstatt das Gefäß mit vielen einzelnen Clippingstrukturen nachzubilden, um es isoliert darzustellen.
Abbildung 26 zeigt ein Beispiel von einem Gefäß mit zwei Aneurysmen, welches sich in unmittelbarer Nähe zu Knochen befindet. In der dreidimensionalen Ansicht rechts ist deutlich zu erkennen, dass Strukturen angezeigt werden, die nicht zum Gefäß gehören (beige). In diesem Fall ist das Gefäß (nachträglich rot eingefärbt) die relevante Struktur und alles andere soll nicht dargestellt werden. Es wird deutlich, dass es für dieses Beispiel keine einfache Clippingstruktur gibt, mit der sich das Gefäß von der restlichen Struktur gut trennen lässt. Auf der rechten Seite ist andeutungsweise eingezeichnet (grün), wie sich die irrelevante Struktur mit mehreren Clippingstrukturen abbilden ließe, um sie anschließend mit inversem Clipping auszublenden.

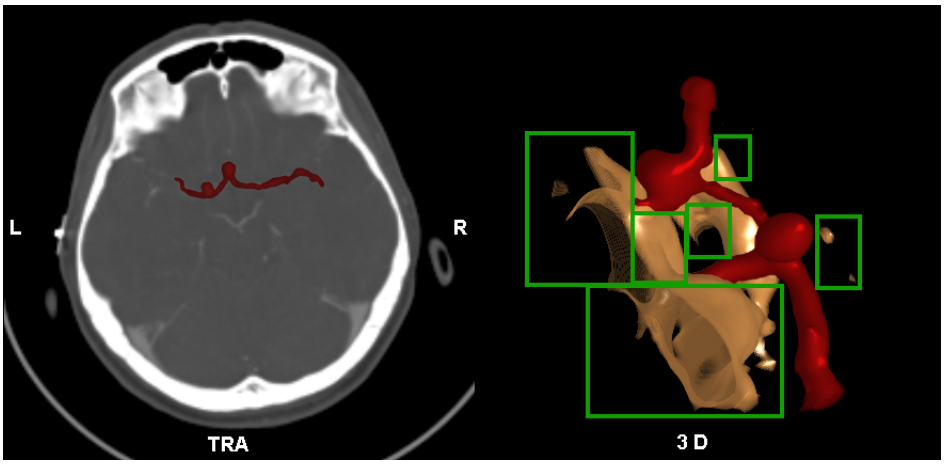


Abbildung 26: Gefäß mit 2 Aneurysmen in Knochennähe (die Gefäße wurden nachträglich mit einem Bildbearbeitungsprogramm rot eingefärbt)

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Für den Benutzer soll eine einfache Option im Segment-Editor hinzukommen, die es ihm erlaubt, zwischen regulärem und inversem Clipping zu wechseln.

3.2.3 Multiple Volumen

Analyse:
In der Volume Rendering Komponente lassen sich beliebig viele Segmente eines Volumens darstellen. Es ist allerdings nicht ohne weiteres möglich, weitere Volumen (mit eigenen Segmenten) hinzuzufügen und zu löschen, obwohl das in der Theorie durchführbar sein sollte (siehe Abbildung 20). An dieser Stelle sei vorweggenommen, dass es für die Darstellung der GPU-basierten Segmentierung unerlässlich ist, mehrere Volumen darzustellen (siehe Kapitel 4.4).

Konzept:
Aus diesem Grund sollen entsprechende Methoden entwickelt werden, um ein Volumen hinzuzufügen und zu entfernen. Dem Dialog zum Erstellen eines neuen Volume Rendering Segments ist ein Button zum Löschen des Volumens hinzuzufügen. Dieser Button soll allerdings nicht für den Planungsdatensatz (das ursprüngliche Volumen) angezeigt werden. Dieses ist als einziges Volumen nicht löschbar.

3.3 GPU-Segmentierung

Die herkömmliche Segmentierung in MOPS 3D bietet verschiedene Hilfsmittel, wie z.B. Segmentierung per Region Growing, über Histogrammschwellwerte, Freihandsegmentierung, Erosion/Dilatation und weitere (siehe Abbildung 18).
Der zeitintensivste Teilschritt ist dabei die Darstellung des Ergebnisses (d.h. alle Voxel, die zur Segmentierung gehören). Weil dies durch Oberflächendarstellung geschieht, muss die Oberfläche des Ergebnisses durch die sog. Triangulation ihrer Voxel angenähert und durch Dreiecke dargestellt werden. Die Triangulation wird durch den *Marching Cubes* Algorithmus realisiert (Näheres dazu gibt es im Paper [LC87] und im Buch [Bir10, S.

290ff.]). Dieser ist jedoch – vor allem bei großen Segmentierungen – sehr zeitintensiv. Die Triangulation und Darstellung einer beispielhaften Segmentierung des Schädelknochens (auf dem MRT Datensatzes eines menschlichen Schädels) benötigt auf der Testmaschine 17 Sekunden.

Deshalb soll ein Segmentierungsverfahren integriert werden, welches auf dem um ein Vielfaches schnelleren Grafikkartenprozessor (GPU) ausgeführt wird anstelle des Hauptprozessors (CPU), wie bei der bisherigen Segmentierung. Der Hauptvorteil eines derartigen Verfahrens besteht darin, dass sich das Ergebnis nach abgeschlossener Segmentierung bereits im Grafikkartenspeicher befindet. Da Raycasting mit Voreen auch auf der Grafikkarte berechnet wird, lässt sich das Ergebnis auf diese Weise mit Volume Rendering in Echtzeit darstellen, weil der Triangulationsschritt entfällt. Ein weiterer Geschwindigkeitszuwachs ist bei der Ausführung des Region Growing Algorithmus zu erwarten, da er auf dem Grafikkartenprozessor stattfindet.

Das zu integrierende Segmentierungsverfahren wird zeitgleich von Marius Wirths entwickelt und wird Region Growing als Hauptsegmentierungsmethode beinhalten.

Es soll eine Schnittstelle konzipiert werden, über die MOPS 3D und das GPU-basierte Segmentierungsverfahren (im Folgenden um der Einfachheit Willen „GPU-Segmentierung“ genannt) miteinander kommunizieren. Diese Schnittstelle soll allgemein gehalten werden, so dass sie nicht speziell auf das zu integrierende Segmentierungsverfahren abgestimmt ist, sondern auch andere passende Implementierungen akzeptiert.

Des Weiteren soll die Benutzeroberfläche für die GPU-Segmentierung nach Vorbild des Dialogs zur herkömmlichen Segmentierung entwickelt werden. Allerdings soll sie nicht so komplex ausfallen wie dieser und in besonderer Hinsicht auf Bedienbarkeit entworfen werden.

3.3.1 Benutzeroberfläche

Im Folgenden wird ein Dialog konzipiert, der zur Steuerung der GPU-Segmentierung dient. Es werden die einzelnen Funktionen beschrieben, die dabei zum Ablauf der Benutzerinteraktion gehören sollen. Abbildung 27 illustriert diesen Ablauf.

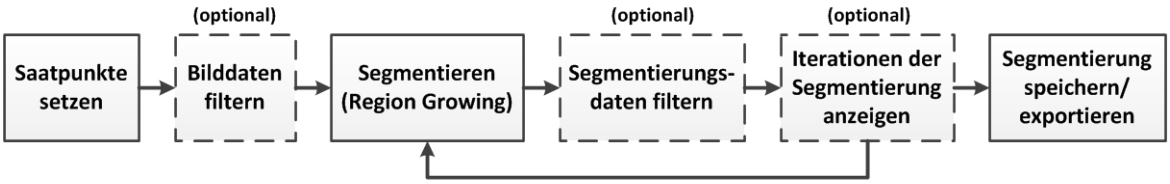


Abbildung 27: Ablauf der Benutzerinteraktion bei der GPU-Segmentierung

Quelle: eigene Darstellung

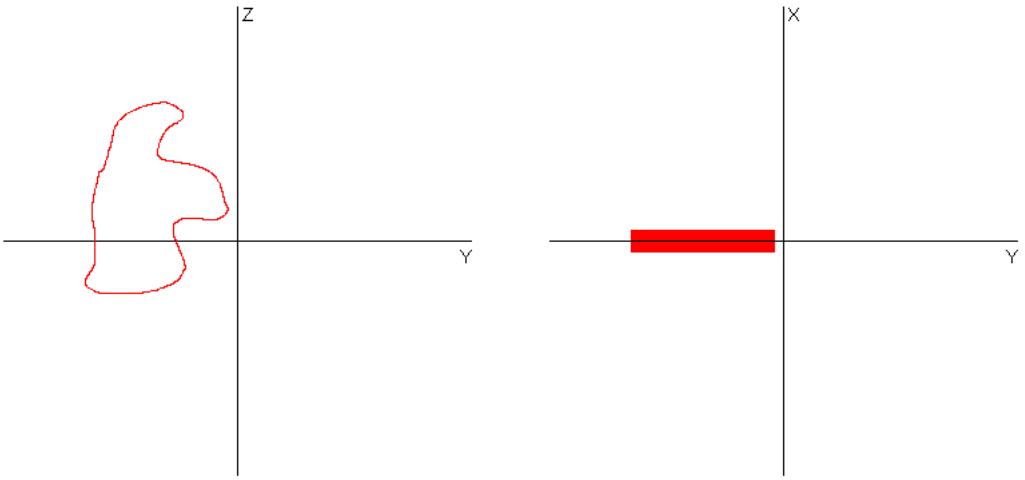
Saatpunke

Die GPU-Segmentierung soll dem Benutzer die Möglichkeit bieten, Region Growing als Segmentierungsmethode zu verwenden. Dieser Algorithmus benötigt zur Ausführung immer mindestens einen Voxel als Startpunkt. Je mehr dieser sog. Saatpunkte festgelegt sind, desto präziser arbeitet der Algorithmus. Saatpunkte können aber auch gegenteilig benutzt werden, um den Algorithmus daran zu hindern, über bestimmte Voxel hinauszuwachsen. Zum besseren Verständnis wird diese Art von Saatpunkt nachfolgend als Nicht-Saatpunkt bezeichnet.

Der Benutzer soll in der Lage sein, Saatpunkte mit der Maus direkt in die 2D Ansicht einzuzeichnen. Damit dies nicht in Konflikt mit der herkömmlichen Maussteuerung steht, soll es im Dialog möglich sein, das Mausverhalten (Mausmodus) entsprechend zu ändern. Man soll Punkte und Linien (Linien entsprechen einer Menge an Saatpunkten) wie in einem Zeichenprogramm frei Hand zeichnen können und bei Bedarf den Anfang und das Ende einer Linie verbinden lassen, um geschlossene Formen zu erhalten.

In einem Zeichenprogramm lässt sich die Pinseldicke festlegen, um breitere Linien zu zeichnen. Im GPU-Segmentierungsdialog soll man die Linienbreite ebenfalls festlegen können. Da es sich hierbei aber um dreidimensionale Linien handelt, die in eine zweidimensionale Schicht der Bilddaten eingezeichnet wird, soll die Linienbreite immer in der Dimension variabel sein, in der gerade nicht gezeichnet wird. Das bedeutet z.B., dass beim Zeichnen in der coronalen Ebene (xz-Ebene) mit einer Linienbreite von 5, jeweils 2 Voxel über und unter den einzelnen Linienpunkten auf der y-Achse eingezeichnet

werden. Abbildung 28 verdeutlicht dies anhand einer beispielhaften Zeichnung aus zwei Sichtweisen.



(a) Zeichnung auf Z-Y-Ebene (b) Dieselbe Zeichnung auf der X-Y-Ebene

Abbildung 28: Freihandzeichnen mit großer Linienbreite

Quelle: eigene Darstellung

Um auch das Zeichnen von Nicht-Saatpunkten zu ermöglichen, sollen alle zuvor genannten Zeichenoperationen bei gedrückter STRG-Taste dieselbe Funktion für Nicht-Saatpunkte bieten. Außerdem sollen Nicht-Saatpunke farblich gekennzeichnet sein, um sie von den Saatpunkten abgrenzen zu können.
Weiterhin sollen alle Zeichnungen[30] rückgängig gemacht werden können.
Abbildung 29 zeigt den Entwurf der Benutzeroberfläche für die Saatpunkte.

[30]Eine Zeichnung bedeutet hierbei alles, was zwischen dem Drücken und Loslassen der Maustaste gezeichnet wird. Beim erneuten Drücken und Loslassen entsteht eine weitere Zeichnung usw.

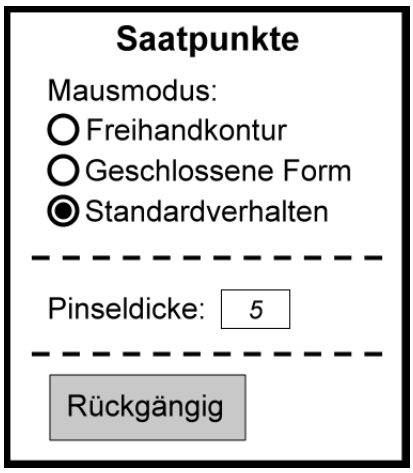


Abbildung 29: Konzept der Benutzeroberfläche der GPU-Segmentierung: Saatpunkte
Quelle: eigene Darstellung

Filtern

Der Benutzer soll die Möglichkeit haben, verschiedene Filter sowohl auf die Bilddaten (vor der Segmentierung) als auch auf die Segmentierungsdaten (d.h. das Ergebnis der Segmentierung) anzuwenden, um z.B. Kanten zu glätten oder Rauschen zu entfernen. Das Filtern der Bilddaten soll zu einer genaueren Segmentierung führen, da zuerst der Filter angewandt wird und auf den gefilterten Bilddaten segmentiert wird. Das Filtern der Segmentierungsdaten dagegen soll zu einer besseren Darstellung des Ergebnisses mit Volume Rendering führen.
Angewandte Filter sollen sich löschen und somit wieder rückgängig machen lassen.
Abbildung 30 zeigt den Entwurf der Benutzeroberfläche für die Filter.

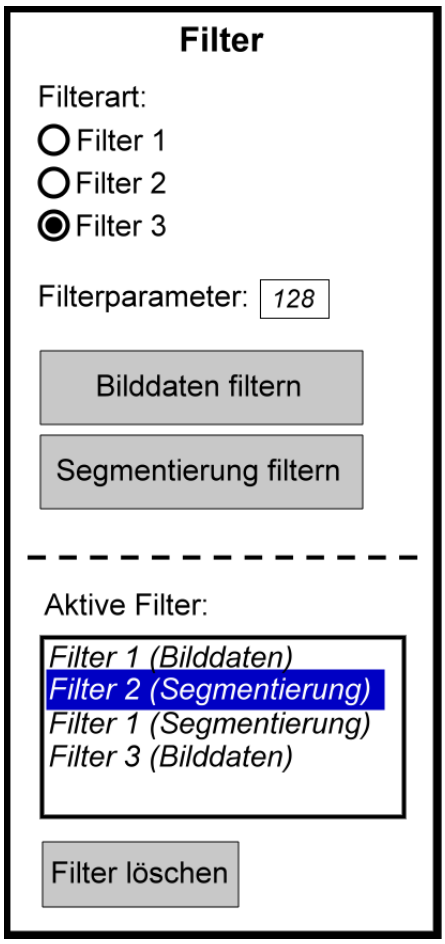


Abbildung 30: Konzept der Benutzeroberfläche der GPU-Segmentierung: Filter

Quelle: eigene Darstellung

Segmentieren, Speichern, Exportieren und Iterationen anzeigen

Der Benutzer soll den Region Growing Algorithmus starten und – im Gegensatz zur herkömmlichen Segmentierung – auch anhalten können. Während die Region durch den Algorithmus wächst, ist diese mit Volume Rendering darzustellen, so dass das dynamische Wachstum visualisiert wird. Dies hat den Vorteil, dass man während des Wachsens der Struktur beobachten kann, wann sie ggf. ausläuft, d.h. Grenzen übertritt, die sie nicht übertreten darf. Nach dem Anhalten oder der ornungsgemäßen Beendigung des Region Growing wird das Ergebnis ebenfalls mit Volume Rendering dargestellt.

Als Eingabeparameter für den Region Growing Algorithmus soll der Benutzer die Schrittweite festlegen können, also wie weit in jede Richtung die benachbarten Voxel bei jeder Iteration überprüft werden. Details zur Funktionsweise und Implementierung eines Region Growing Algorithmus gibt es im Buch [GWE04] und im Paper [AB94].
Eine weitere Hilfestellung für den Benutzer soll die Anzeige der einzelnen Iterationen sein, mit der sich alle Iterationen nach abgeschlossener oder angehaltener Segmentierung einzeln durchblättern lassen. Außerdem ist es erforderlich, das Region Growing ab einer ausgewählten Iteration starten zu können, so dass nach einer kleinen Korrektur nicht von vorne begonnen werden muss.
Abbildung 31 zeigt den Entwurf der Benutzeroberfläche für das Segmentieren und die Iterationsanzeige/-auswahl.

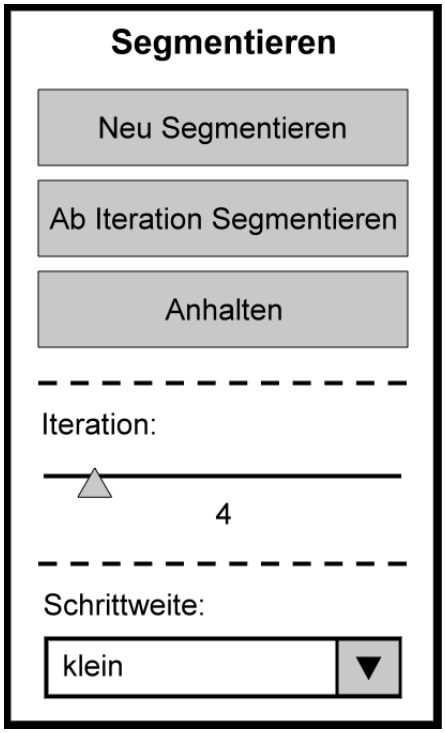


Abbildung 31: Konzept der Benutzeroberfläche der GPU-Segmentierung: Segmentieren
Quelle: eigene Darstellung

Ist die Segmentierung abgeschlossen (oder angehalten worden), soll sie als Volumen abgespeichert werden können. Des Weiteren soll es möglich sein, das Ergebnis der GPU-

Segmentierung in den Dialog der herkömmlichen Segmentierung zu exportieren. Dort lässt sich die Segmentierung dann mit diversen Funktionen feinjustieren, die es bei der GPU-Segmentierung nicht gibt (weil es keinen entscheidenden Geschwindigkeitsvorteil brächte, diese auf die GPU auszulagern). Der herkömmliche Segmentierungsdialog soll ebenfalls um die Funktion erweitert werden, die Segmentierung (wieder zurück) in den GPU-Segmentierungsdialog zu exportieren.
Abbildung 32 zeigt den Entwurf der Benutzeroberfläche für das Speichern und Exportieren.

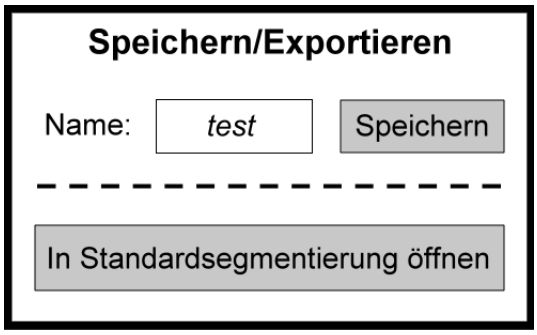


Abbildung 32: Konzept der Benutzeroberfläche der GPU-Segmentierung: Speichern
Quelle: eigene Darstellung

3.3.2 Schnittstelle

MOPS 3D soll mit der GPU-Segmentierung über eine vordefinierte Schnittstelle kommunizieren. Die GPU-Segmentierung wird dabei eine Reihe von Schnittstellenmethoden zur Verfügung stellen, die in der Arbeit von Marius Wirths [Wir12] beschrieben werden, während an dieser Stelle primär die Schnittstellenmethoden vorgestellt werden, die von MOPS 3D bereitgestellt werden müssen. Allerdings werden einige Schnittstellenmethode der GPU-Segmentierung zum besseren Verständnis der Kommunikation auch hier erwähnt. Nachfolgend werden die benötigten Funktionen einzeln vorgestellt und abschließend in einem umfassenden Diagramm dargestellt.
Um die GPU-Segmentierung überhaupt zu ermöglichen, muss MOPS 3D in der Lage sein, sie zu instanziieren. Deshalb muss die GPU-Segmentierung Methoden zur Initiali-

sierung und Deinitialisierung bereitstellen, wie in Abbildung 33 dargestellt.

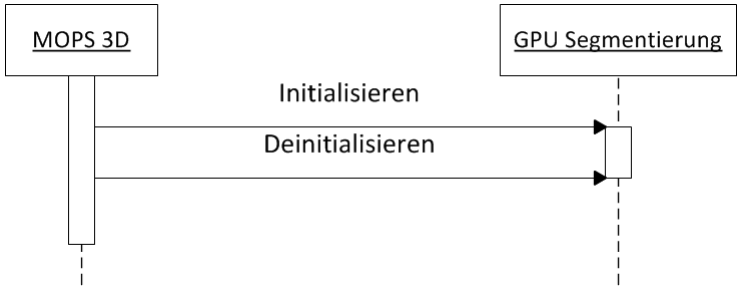


Abbildung 33: (De-)Initialisierungsmethoden der Schnittstelle zur GPU-Segmentierung

Quelle: eigene Darstellung

Wie bereits im vorherigen Kapitel beschrieben, benötigt die GPU-Segmentierung mindestens einen Saatpunkt für das Region Growing. MOPS 3D muss der GPU-Segmentierung also neue Saatpunkte mitteilen können und diese ggf. auch wieder entfernen lassen. Die Methoden zum Hinzufügen und Entfernen von Filtern funktionieren analog dazu. Abbildung 34 fasst die Methoden für Saatpunkte und Filter zusammen.

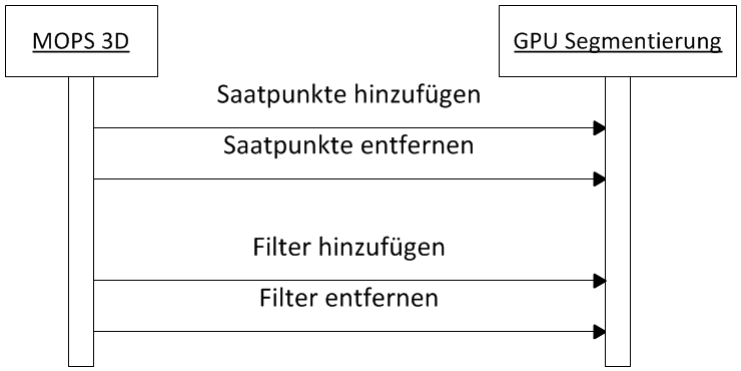


Abbildung 34: Saatpunkt- und Filtermethoden der Schnittstelle zur GPU-Segmentierung

Quelle: eigene Darstellung

Weitere Methoden sind erforderlich zum Starten und Anhalten der Segmentierung. Außerdem soll das Region Growing asynchron erfolgen, damit die Einzelschritte in Echtzeit

dargestellt werden können. Aus diesem Grund muss die GPU-Segmentierung MOPS 3D den Status der Segmentierung übermitteln können, d.h. er zeigt an, ob die Segmentierung gerade läuft oder bereits abgeschlossen ist. Nach jedem Einzelschritt soll MOPS 3D weiterhin dazu aufgefordert werden, die Anzeige zu aktualisieren. In Abbildung 35 werden die Methoden zur Segmentierung verdeutlicht.

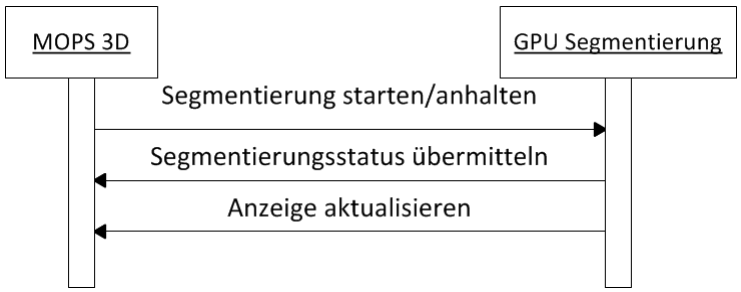


Abbildung 35: Segmentierungsmethoden der Schnittstelle zur GPU-Segmentierung
Quelle: eigene Darstellung

Zur Darstellung ihrer Ergebnisse benötigt die GPU-Segmentierung einige Informationen von MOPS 3D. Zum besseren Verständnis der Notwendigkeit dieser Informationen wird zunächst erklärt, wie die GPU-Segmentierung ihre Ergebnisse visualisieren soll:
Für die GPU-Segmentierung wird zunächst eine Kopie des Planungsdatensatzes erzeugt und in das bestehende Voreen Netzwerk eingebunden, so dass es mit Volume Rendering dargestellt wird. Nun kann die GPU-Segmentierung dieses Volumen direkt im Grafikkartenspeicher manipulieren, um es mit ihrem Ergebnis zu überschreiben. Für diese Manipulation muss der GPU-Segmentierung die Größe des Volumens und seine Adresse im Grafikkartenspeicher bekannt sein. Mit Hilfe von OpenGL lässt sich das Volumen als 3D Textur speichern, so dass es ausreicht, die ID dieser Textur anstelle der Speicheradresse an die GPU-Segmentierung zu übergeben. Um die 3D Textur auslesen zu können, benötigt die GPU-Segmentierung zusätzlich noch den OpenGL Rendering Context. Abbildung 36 verdeutlicht diesen Datenaustausch.

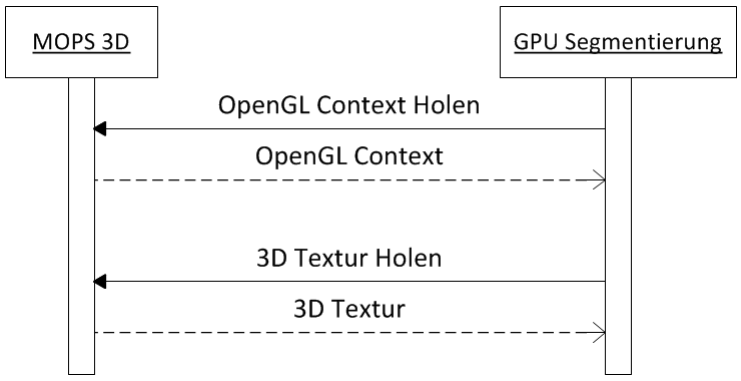


Abbildung 36: Wichtige OpenGL-Methoden der Schnittstelle zur GPU-Segmentierung
Quelle: eigene Darstellung

Der geteilte Zugriff von MOPS 3D und der GPU-Segmentierung auf OpenGL macht ein Synchronisierungsverfahren erforderlich, da sonst unvorhersehbares oder fehlerhaftes Verhalten auftreten kann. Ein binärer Semaphor ist hierfür ausreichend, also eine Methode, die dem Aufrufer (entweder MOPS 3D oder der GPU-Segmentierung) exklusiven Zugriff auf OpenGL gewährt und eine Methode, die diesen Zugriff wieder freigibt. Abbildung 37 stellt diesen Zusammenhang dar.

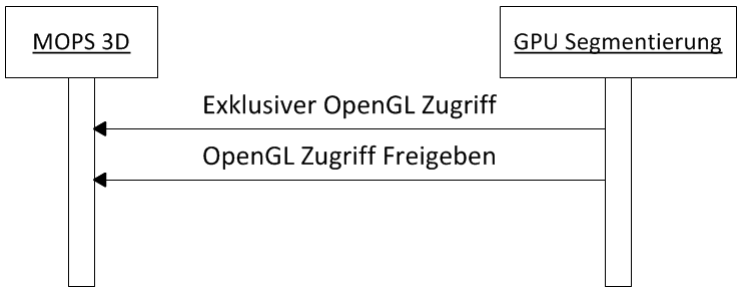


Abbildung 37: Semaphormethoden der Schnittstelle zur GPU-Segmentierung
Quelle: eigene Darstellung

Nach abgeschlossener oder angehaltener Segmentierung muss MOPS 3D in der Lage sein, die Anzahl der Iterationen zu erhalten, um dem Benutzer diese Information zu präsentieren. Weiterhin soll der Benutzer eine bestimmte Iteration anzeigen können, demnach ist es erforderlich, auch hierfür eine Methode anzulegen. Abbildung 38 verdeutlicht diesen

Sachverhalt.

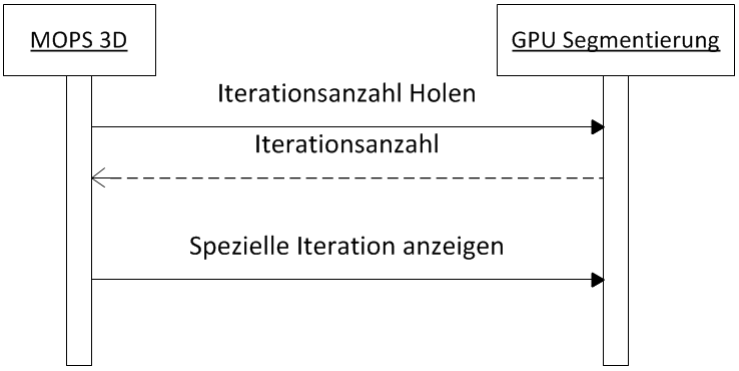


Abbildung 38: Iterationsmethoden der Schnittstelle zur GPU-Segmentierung
Quelle: eigene Darstellung

Möchte der Benutzer das Ergebnis der GPU-Segmentierung in der herkömmlichen Segmentierung darstellen oder umgekehrt, so soll es als Bytemaske übergeben werden, um nicht unnötig Speicherplatz zu beanspruchen. Das bedeutet, jedes Byte in der Maske entspricht einem Voxel im Volumen und beinhaltet eine Eins falls der Voxel Teil der Segmentierung ist, oder andernfalls eine Null. Entsprechende Methoden zum Holen oder Setzen der Maske sind demnach erforderlich, wie in Abbildung 39 dargestellt.

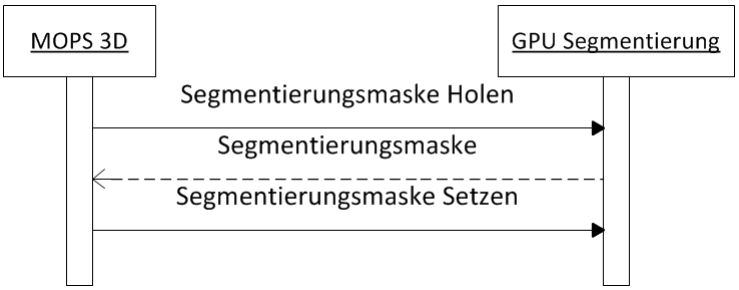


Abbildung 39: Import-/Exportmethoden der Schnittstelle zur GPU-Segmentierung
Quelle: eigene Darstellung

Abbildung 40 fasst alle benötigten Schnittstellenfunktionen in einem Diagramm zusammen.

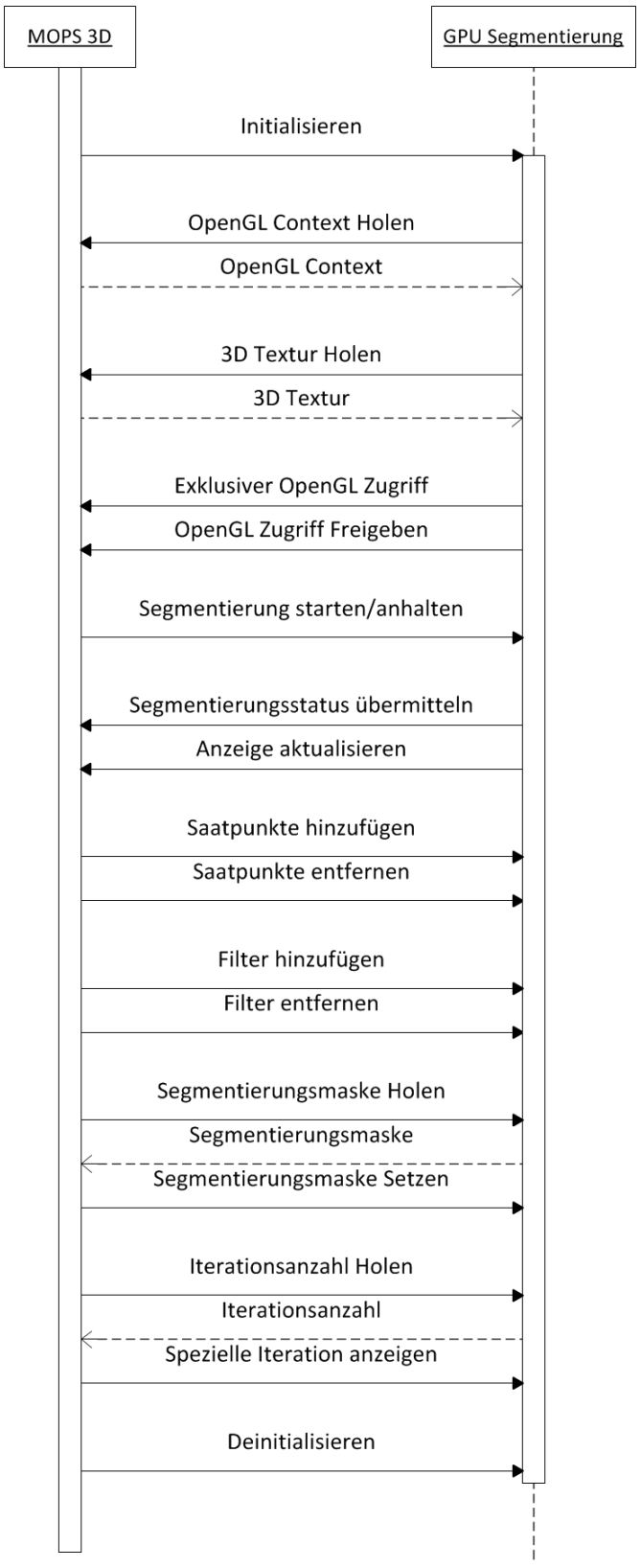


Abbildung 40: Gesamte Schnittstelle zur GPU-Segmentierung

Quelle: eigene Darstellung

4 Implementierung

Im Folgenden wird die Implementierung der zuvor konzipierten Entwürfe vorgestellt. Im Vordergrund stehen dabei die Aspekte, die von zentraler Bedeutung für die Entwicklung sind, sowie die Probleme, die sich bei der Implementierung ergeben.

4.1 Integration der Voreen Version 2.6.1

Da Voreen (in der Version r13) bereits in MOPS 3D integriert ist, ist die Annahme naheliegend, dass die Integration der neuen Version sich als einfach gestaltet, da sich viele der Strukturen, die der Integration von Voreen r13 dienen, wahrscheinlich auch für die Integration von Voreen 2.6.1 wiederverwenden lassen. Mit der Veröffentlichung der Version 2.0 änderte sich die Architektur von Voreen allerdings grundlegend. In der Übersicht der Neuerungen (Changelog, siehe [Vora]) lassen sich die Änderungen an Voreen identifizieren, die bei der Integration in MOPS 3D potentielle Probleme darstellen. Jede dieser Änderungen bringt gewisse Schwierigkeiten bei der Integration mit sich. Diese werden nachfolgend im Einzelnen vorgestellt.

4.1.1 Ersetzen des Nachrichtensystems

Bisher werden Änderungen an Parametern, wie z.B. der Transferfunktion, der Segmentfarbe oder Einstellungen zum Clipping, vom Dialog in MOPS 3D an Voreen mit Hilfe eines Nachrichtensystems übermittelt. Eine Nachricht besteht dabei aus einem Identifier und einem Wert. Der Identifier bezeichnet die Variable, die sich ändern soll und kann mit einer Zeichenkette angegeben werden. Voreen leitet diese Nachricht an die gewünschten

Prozessoren weiter, wo dann der Wert der Nachricht die entsprechende Variable überschreibt. Für jede Art von Parameter (Wert) existiert ein eigener Nachrichtentyp (Bool, String, Integer, etc.). Listing 2 verdeutlicht das Nachrichtensystem am Beispiel von Clippingebenen. Die Ebenen werden hier der Größe des Volumens angepasst (0% und 100%) und das Clipping wird aktiviert („UseClipping“).

```
MsgDstr.postMessage(new voreen::IntMsg("set.RightClipPlane", 0));
MsgDstr.postMessage(new voreen::IntMsg("set.LeftClipPlane", 100));
MsgDstr.postMessage(new voreen::IntMsg("set.BackClipPlane", 0));
MsgDstr.postMessage(new voreen::IntMsg("set.FrontClipPlane", 100));
MsgDstr.postMessage(new voreen::IntMsg("set.BottomClipPlane", 0));
MsgDstr.postMessage(new voreen::IntMsg("set.TopClipPlane", 100));
MsgDstr.postMessage(new voreen::BoolMsg("set.UseClipping", true));
```

Listing 2: Ändern der Clippingeigenschaften mit dem Nachrichtensystem (Voreen r13)

Seit Voreen 2.0 existiert dieses Nachrichtensystem nicht mehr und wurde durch eine andere Funktionalität ersetzt: Anstelle von Identifiers besitzt jeder Prozessor nun verschiedene Properties (Eigenschaften), die jeweils einen Namen und Wert besitzen. Es gibt verschiedene Typen von Properties, um alle Arten von Werten abzudecken (Bool, String, Integer, etc.). Der Vorteil gegenüber den Identifiers liegt darin, dass Properties zentralisiert funktionieren (d.h. sie vereinen Namen und Wert in ihrer Struktur) und dass sie Werte auch validieren können (z.B. kann bei Integer Properties ein Minimal- und Maximalwert angegeben werden).
Mit dem neuen Properties-System müssen die Prozessoren direkt angesprochen werden, da es keinen zentralen Nachrichtenempfänger mehr gibt, der Änderungen weiterleitet. Listing 3 verdeutlicht das Properties-System am gleichen Beispiel wie Listing 2. Der Prozessor muss hierbei bekannt sein, da er direkt angesprochen wird.

```
Processor *proxyGeometry; //muss den zu veraendernden Prozessor beinhalten
proxyGeometry->getProperty("rightClippingPlane")->set(0);
proxyGeometry->getProperty("leftClippingPlane")->set(100);
proxyGeometry->getProperty("backClippingPlane")->set(0);
proxyGeometry->getProperty("frontClippingPlane")->set(100);
proxyGeometry->getProperty("bottomClippingPlane")->set(0);
proxyGeometry->getProperty("topClippingPlane")->set(100);
proxyGeometry->getProperty("useClipping")->set(true);
```

Listing 3: Ändern der Clippingeigenschaften mit dem Properties-System (Voreen 2.0)

Eine weitere Neuerung von Version 2.0 ist die Tatsache, dass Properties miteinander verknüpft (sog. Linking) werden können. Ändert sich ein Property, das mit einem weiteren Property verknüpft ist, so ändert sich das andere Property automatisch mit. Dies ist vor allem nützlich, um das Verhalten von Prozessoren miteinander abzugleichen, z.B. Kamera- und Clippingeigenschaften bei mehreren Segmenten, d.h. bei mehreren Prozessorteilbäumen (siehe unten).
Ein Problem, das durch die Umstellung auf das Properties-System in MOPS 3D entsteht, betrifft die Kamera. Bei jeder Änderung der Kamera in MOPS 3D wird auch die Kamera in Voreen aktualisiert, damit Translation, Rotation und Skalierung synchron sind. Aktualisiert man nun die Voreen-Kamera über das entsprechende Property – nämlich *CameraProperty* –, so werden nicht alle Kameraeigenschaften übernommen. Die *set()* Methode des *CameraProperty* erwartet eine Referenz auf ein *Camera* Objekt. Von dieser werden dann Position, Fokus und der Vektor übergeben, der nach „oben“ zeigt. Eigenschaften, wie z.B. der Darstellungsbereich (d.h. ob es sich um eine perspektivische oder orthogonale Projektion handelt) werden dabei vernachlässigt. Aus diesem Grund muss die *set()* Methode insofern angepasst werden, dass sie das *Camera* Objekt komplett kopiert. Listing 4 und 5 zeigen die unveränderte und die angepasste *set()* Methode. Die Variable *value_* steht dabei für das Kamera Objekt, das von *CameraProperty* repräsentiert wird.

```
void CameraProperty::set(const tgt::Camera& camera) {
    if (value_.getPosition()   !=  camera.getPosition()   ||
        value_.getFocus()      !=  camera.getFocus()      ||
        value_.getUpVector()   !=  camera.getUpVector())
    {
        value_.setPosition(camera.getPosition());
        value_.setFocus(camera.getFocus());
        value_.setUpVector(camera.getUpVector());
    }
}
```

Listing 4: Unveränderte set() Methode des CameraProperty

```
void CameraProperty::set(const tgt::Camera& camera) {
    value_ = camera;
}
```

Listing 5: Angepasste set() Methode des CameraProperty

4.1.2 Entfernen des Combine Prozessors

Ab Version 2.0 gibt es in Voreen den Combine Prozessor nicht mehr. Dieser wird benötigt, um mehrere 2D Bilder miteinander zu einem Ausgabebild zu verschmelzen. In MOPS 3D wird er verwendet, um die Ausgabebilder der einzelnen Prozessorteilbäume (d.h. der verschiedenen Segmente) zu einem Bild zusammenzufassen, welches dann auf dem Canvas Prozessor angezeigt werden kann. Es gibt zwar einen Compositor Prozessor in Voreen 2.0, der imstande ist, zwei Bilder mit Blending zu einem zusammenzuführen, allerdings beschränkt sich die Anzahl der möglichen Eingangsbilder auf zwei. Ersetzte man also den Combine Prozessor in MOPS 3D durch den Compositor Prozessor, führte das zu dem Problem, dass sich nicht mehr als zwei Segmente anzeigen lassen.
Nach Rücksprache mit den Entwicklern (auf der Voreen Mailing List, siehe [Vorb]) kam ich zu dem Entschluss, einen eigenen Prozessor – nach Maßstab des Compositor Prozessors – zu entwerfen, der imstande ist, beliebig viele Eingabebilder zu akzeptieren und mit Hilfe von Blending genau ein Ausgabebild liefert.
Der Prozessor soll den Namen „MultiCompositor“ tragen und als neues Voreen-Modul hinzugefügt werden, um zukünftige Entwicklung und die Integration zukünftiger Voreen

Versionen zu erleichtern. Module in Voreen sind in sich geschlossene Strukturen, die z.B. neue Prozessoren oder Bibliotheken enthalten können. Sie dienen der Erweiterung von Voreen und können beliebig aktiviert und deaktiviert werden (durch das Deaktivieren unbenötigter Module lässt sich z.B. Speicherplatz einsparen). Anhang D bietet einen Überblick, wie sich Module in die Voreen Architektur eingliedern.
Nach Anleitung ([Vord, S. 31–50]) lässt sich schnell ein Modul namens „mops“ samt neuem Prozessor hinzufügen. Der Prozessor wird nach dem Vorbild des Compositor Prozessors entworfen, mit dem Unterschied, dass der Inport multiple Verbindungen (d.h. mehrere Eingangsbilder) akzeptiert. In der Hauptmethode des Prozessors werden alle Eingangsbilder dann sequentiell paarweise abgearbeitet. D.h. die ersten zwei Bilder werden vermischt und das Ergebnis zwischengespeichert. Dieses wird dann mit dem dritten Bild vermischt und wieder temporär gespeichert, woraufhin das neue Ergebnis mit dem vierten Bild vermischt wird usw. Das tatsächliche Vermischen zweier Bilder wird durch einen Shader realisiert, der einfaches Alpha Blending auf die Pixel anwendet. Listing 6 zeigt den vereinfachten Shader Code für das Alpha Blending. Die Variablen *color0* und *color1*, sowie *depth0* und *depth1* stehen dabei jeweils für die beiden zu vermischenden Farb- und Tiefenwerte.

```
if(onlyUseFirstImage_)
{
    //Nur ein Eingangsbild, also Werte unveraendert uebernehmen
    FragData0 = color0;
    gl_FragDepth = depth0;
}
else
{
    if(depth0 < depth1)
    {
        FragData0 = color1 * (1.0 - color0.a) + color0 * color0.a;
        gl_FragDepth = depth0;
    }
    else
    {
        FragData0 = color0 * (1.0 - color1.a) + color1 * color1.a;
        gl_FragDepth = depth1;
    }
}
```

Listing 6: Shader Code für Alpha Blending

Listing 7 zeigt den Code der Hauptmethode des MultiCompositor Prozessors und veranschaulicht das paarweise Abarbeiten der Eingangsbilder. Die Variable *inports* enthält dabei die Eingangsbilder. *source*, *target* und *temp* repräsentieren Texturen.

```
source = inports[0]; //erstes Eingangsbild

//inports.size() ist die Anzahl der Eingangsbilder
for(int i=0; i<inports.size(); i++)
{
    //Erstes zu mischendes Bild
    shader->setUniform("texture0", source);

    //Ergebnis auf den Outport legen
    target = outport;

    if(inports.size() == 1)
    {
        //Es gibt nur ein Eingangsbild
        shader->setUniform("onlyUseFirstImage_", true);
    }
    else
    {
        shader->setUniform("onlyUseFirstImage_", false);

        //Zweites zu mischendes Bild
        shader->setUniform("texture1", inports[i]);

        //Ergebnis zwischenspeichern, falls dies nicht
        //das letzte Eingangsbild ist
        if(i < inports.size() - 1)
            target = temp;
    }

    shader->run(); //Ergebnis auf der Zieltextur rendern

    source = target; //Naechstes Eingangsbild abarbeiten
}
```

Listing 7: Sequentiell paarweise Abarbeitung der Eingangsbilder des MultiCompositor Prozessors

Einer der Vorzüge dieser Implementierung ist, dass sie auch funktioniert, wenn es nur ein Eingangsbild gibt – im Gegensatz zum Combine Prozessor. Denn der Combine Prozessor wird immer nur dann in das Prozessornetzwerk eingefügt, wenn es mehr als einen

Prozessorteilbaum (also mehrere Segmente) gibt, und wieder entfernt, falls es wieder nur einen gibt. Der MultiCompositor Prozessor dagegen kann einfach fester Bestandteil des Netzwerks bleiben.

4.1.3 Änderungen mit geringer Auswirkung

Eine weitere Neuerung in Voreen 2.0 ist die veränderte Funktionalität des Qualitätsfaktors des Raycasters. In Version r13 lässt sich die Darstellungsqualität eines Raycasters über den Qualitätsfaktor als Parameter steuern. Dazu gibt es sechs vordefinierte Qualitätseinstellungen (von sehr niedrig bis sehr hoch) zwischen denen man wählen kann. Jede dieser Voreinstellungen steht intern für einen numerischen Wert, aus dem dann die Abtastrate berechnet wird. Die Abtastrate ist ein Distanzmaß für die Schrittweite des Suchstrahls beim Raycasting, d.h. wie weit die abgetasteten Punkte der einzelnen Iterationen auseinander liegen. Je enger sie beisammen liegen, desto höher ist die resultierende Darstellungsqualität. Ab Version 2.0 existiert dieser Parameter nicht mehr und wird abgelöst durch einen Parameter namens „Samplingrate“ (Abtastrate). Im Gegensatz zum Qualitätsfaktor ist sie ein numerischer Wert. Aus diesem wird die Abtastrate zwar auch intern errechnet, die Umrechnung erfolgt jedoch auf andere Weise (siehe Kapitel 5.1). Um diese Änderung in MOPS 3D zu integrieren wird der globale Volume Rendering Konfigurationsdialog (siehe Abbildung 19) insofern angepasst, dass die Liste mit vordefinierten Qualitätseinstellungen durch einen Schieberegler ersetzt wird, der die Samplingrate auf einer Skala zwischen 0.01 und 20 reguliert (der Minimal- und Maximalwert sind übernommen aus der VoreenVE Anwendung).

Weiterhin wird die Interaction Coarseness[31] (siehe Kapitel 2.3) ab Version 2.0 nicht mehr in einem eigenen Prozessor (CoarsenessRenderer) gesteuert. Stattdessen ist sie direkt im Raycaster integriert und wird automatisch bei Interaktion aktiviert. Bei Voreen wird Interaktion allerdings über ein spezielles Ereignissystem gesteuert, so dass die Interaction

[31] Die Verschlechterung der Bildqualität während der (Maus-)Interaktion zugunsten einer flüssigeren Darstellung. Interaction Coarseness bezeichnet weiterhin auch den gleichnamigen Parameter, der die Intensität dieser Verschlechterung steuert.

Coarseness in MOPS 3D nicht von alleine ausgelöst wird, da MOPS 3D Interaktion über das MFC Ereignissystem steuert. Daher ist es nötig, bei Anfang und Ende einer Interaktion in MOPS 3D (z.B. Drücken und Loslassen der Maustaste während einer Rotation) den Raycaster in den Interaktionsmodus zu versetzen bzw. diesen wieder zu lösen. Erst dann verhält sich die Interaction Coarseness wieder korrekt.

Das Laden von Volumen hat sich ebenfalls geändert. Bisher wurden dafür die Prozessoren *VolumeSetSource* und *VolumeSelector* benötigt. Der erste, um eine Menge an Volumen zu laden und zu verwalten, der zweite, um eins der Volumen in der Menge auszuwählen. Selbst für ein einziges Volumen war man auf diese zwei Prozessoren angewiesen. Seit Version 2.0 repräsentiert der *VolumeSource* Prozessor genau ein Volumen, welches er auch einlädt. Entsprechend werden *VolumeSetSource* und *VolumeSelector* im Prozessornetzwerk einfach durch *VolumeSource* ersetzt.
Außerdem hat sich das Initialisieren von Voreen 2.0 insofern verändert, dass die wichtigen Strukturen wie der *NetworkEvaluator* und der *Canvas* Prozessor einen sog. geteilten Kontext zu ihrer Initialisierung benötigen. Dieser geteilte Kontext fungiert als Schnittstelle zum Framework der Benutzeroberfläche (in diesem Fall MFC) und bietet Voreen dadurch Zugriff auf Funktionen, die z.B. die Bildschirmausgabe erneuern oder Events wie Maussteuerung weiterleiten (im Falle von MOPS 3D allerdings nicht benötigt). Diese Schnittstelle liegt in Voreen 2.0 bereits als Basisklasse und als Spezialfall für das Framework Qt[32] vor. Es wird jedoch auch eine Ableitung für MOPS 3D (d.h. für MFC) benötigt. Allerdings beschränkt sich diese Ableitung auf das Überschreiben einiger weniger Methoden, die eines Aufrufs zur Bildschirmausgabenerneuerung bedürfen.

[32]http://qt.nokia.com

4.2 Erweiterung des Clipping

4.2.1 Form der Clippingstruktur

Das Clipping der Volume Rendering Komponente wird – wie bereits in Kapitel 3.2.1 erwähnt – um die Möglichkeit erweitert, andere Begrenzungsstrukturen zu verwenden als den herkömmlichen Quader. Es wird beispielhaft ein Ellipsoid als Alternative implementiert. Dabei wird so vorgegangen, dass weitere Strukturen mit möglichst geringem Aufwand hinzugefügt werden können.
Es wird zunächst erklärt, wie Clipping mit einem Quader funktioniert und dann auf Clipping mit einem Ellipsoid eingegangen. Der CubeMeshProxyGeometry Prozessor erstellt anhand der Clippingparameter ein sog. Mesh eines Quaders, d.h. eine Datenstruktur, die alle Punkte (Vertices) und Seiten des Quaders enthält. Dieses Mesh beinhaltet bereits die Texturkoordinaten, welche – genau wie die Koordinaten der Vertices – angepasst wurden an die angegebenen Clippingparameter, so dass der Quader genau dem gewünschten Clippingbereich entspricht. Dies hat den Effekt, dass nur der Bereich dargestellt wird, der sich in dem Quader befindet.
Der Versuch, dieses Verfahren für ein Ellipsoid anzuwenden, gestaltet sich allerdings als nicht sehr zufriedenstellend, da ein Ellipsoid ein weitaus komplizierterer geometrischer Körper ist, als ein Quader. Das Problem liegt darin, dass er eine runde Form hat und nicht ohne weiteres durch Seiten beschrieben werden kann, wie ein Quader. Entweder müsste man Genauigkeit opfern und die rundliche Form durch wenige Dreiecke annähern oder man verwendet sehr viele Dreiecke, was zwar zu einer rundlicheren und genaueren Darstellung führt, allerdings auch sehr ressourcenintensiv ist.
Da beide Ansätze schlechte Kompromisse sind, wird die Problematik im Fragment Shader des Raycaster Prozessors gelöst. Dort wird ohnehin schon für jeden Pixel (also für jeden Suchstrahl) jeder Voxel entlang des Suchstrahls abgetastet, so dass es einfach wäre, dort abzufragen, ob der Voxel sich innerhalb des Ellipsoids befindet oder nicht. Die

Gleichung eines Ellipsoids lautet:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = 0 \tag{4.1}$$

a, b und c stehen dabei für die Längen der Halbachsen, also sozusagen der drei verschiedenen Radien. Alle Punkte, die innerhalb eines Ellipsoids liegen, erfüllen demnach folgende Bedingung:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} \leq 1 \tag{4.2}$$

Liegt ein Voxel außerhalb des Ellipsoids, soll es nicht dargestellt werden, d.h. sein Beitrag zur Ausgabe muss verhindert werden. Dies geschieht, indem man ihn vollständig transparent macht, indem man seinen Alpha-Farbwert auf Null setzt. Listing 8 zeigt, wie der GLSL Code aussieht, der dies bewerkstelligt.

```
void ellipsoidClipping(in vec3 voxel, inout float alpha)
{
    //Koordinaten transformieren
    vec3 pos = voxel - center_;

    //Verwirf (d.h. alpha=0) alle Punkte, die ausserhalb des
    //Ellipsoids liegen
    if((pos.x * pos.x) / (radius_.x * radius_.x) +
       (pos.y * pos.y) / (radius_.y * radius_.y) +
       (pos.z * pos.z) / (radius_.z * radius_.z) > 1.0)
        alpha = 0.0;
}
```

Listing 8: Clippingmethode für Ellipsoide für den Fragment Shader des Raycasters

Die Variable *pos* stellt dabei die Position des zu überprüfenden Voxels dar und *radius_* die Längen der drei Halbachsen. Die Koordinaten der Voxel liegen zwischen [0, 0, 0] und [1, 1, 1]. Aus diesem Grund wird die Mitte des Ellipsoids (die Variable *center_*) erst vom zu überprüfenden Voxel abgezogen, um ihn zu transformieren. Der Raycaster Prozessor beinhaltet eine Variable namens *clipGeometry_*, die dem Shader mitteilt, welche Clippingmethode er anwenden soll (Quader oder Ellipsoid).
Die Methode wird in einem eigenen Shader implementiert und ebenfalls in das Modul

„mops“ eingepflegt. Auf diese Weise lassen sich in Zukunft mit Leichtigkeit weitere Clippingstrukturen einfügen, indem man den Shader um eine weitere Methode vergrößert. Diese muss überprüfen, ob ein Voxel in der gewünschten Struktur liegt oder nicht und ggf. seinen Alphawert auf Null setzt. Im Shader des Raycasters muss der Aufruf dieser Methode dann anhand der *clipGeometry_* Variable erfolgen.

4.2.2 Inverses Clipping

Die Ermöglichung von inversem Clipping nach dem Konzept von Kapitel 3.2.2 setzt direkt am Lösungsansatz der ellipsoiden Clippingstruktur an. Dort wird im Raycaster Fragment Shader ermittelt, welche Voxel innerhalb der Struktur liegen und für alle anderen wird der Alphawert auf Null gesetzt. D.h. um das Clipping zu invertieren, müssen lediglich die Alphawerte der Voxel, die innerhalb der Struktur liegen, auf Null gesetzt werden und alle anderen bleiben erhalten. Listing 9 zeigt den modifizierten GLSL Code des Clipping für Ellipsoide.

```
void ellipsoidClipping(in vec3 voxel, inout float alpha, in bool negative)
{
    //Koordinaten transformieren
    vec3 pos = voxel - center_;

    if(negative)
    {
        if((pos.x * pos.x) / (radius_.x * radius_.x) +
           (pos.y * pos.y) / (radius_.y * radius_.y) +
           (pos.z * pos.z) / (radius_.z * radius_.z) <= 1.0)
            alpha = 0.0;
    }
    else
    {
        if((pos.x * pos.x) / (radius_.x * radius_.x) +
           (pos.y * pos.y) / (radius_.y * radius_.y) +
           (pos.z * pos.z) / (radius_.z * radius_.z) > 1.0)
            alpha = 0.0;
        }
}
```

Listing 9: Modifizierte Clippingmethode für Ellipsoide für inverses Clipping

Die Methode erhält einen neuen Paramater (*negative*), der die Information enthält, ob

das Clipping invertiert werden soll oder nicht.

Für alle im Shader hinzugefügten Clippingstrukturen lässt sich das Clipping auf diese Weise invertieren. Die Ausnahme bildet das herkömmliche Clipping für Quader, da die Vorgehensweise dort anders ist (siehe Kapitel 4.2.1). Würde man versuchen, inverses Clipping für Quader auf die gleiche Art und Weise zu erreichen, müsste man ein Mesh für einen hohlen Quader generieren, also das Gegenstück. Das stößt zwar nicht auf die Probleme, die bei rundlichen Strukturen auftreten, aber es ist vergleichsweise einfacher, das inverse Clipping für Quader auch im Raycaster Shader zu realisieren. Dank der Geschwindigkeit, die mit der GPU möglich ist, sollte dies auch nicht zu einem signifikanten Performanzunterschied führen. Das Vorgehen ist hierbei wieder das gleiche: In einer eigenen Methode wird überprüft, ob der Voxel innerhalb des Quaders liegt und falls ja, wird sein Beitrag zur Ausgabe verhindert. Listing 10 zeigt den dafür erforderlichen GLSL Code.

```
void boxClippingNegative(in vec3 voxel, inout float alpha)
{
    //Koordinaten transformieren
    vec3 pos = voxel - center_;

    //Verwirf (d.h. alpha=0) alle Punkte, die innerhalb des
    //Quaders liegen
    if((pos.x <= radius_.x && pos.x >= -radius_.x) &&
       (pos.y <= radius_.y && pos.y >= -radius_.y) &&
       (pos.z <= radius_.z && pos.z >= -radius_.z))
        alpha = 0.0;
}
```

Listing 10: Inverse Clippingmethode für Quader für den Raycaster Fragment Shader

Die Variable *radius_* enthält dabei die Länge der Halbachsen des Quaders, genau wie beim Ellipsoid[33]. Die Methode befindet sich im gleichen Shader wie die Methode für das Clipping von Ellipsoiden im „mops“ Modul. Im Raycaster Shader wird dann anhand der zwei Variablen *clipGeometry_* und *clipNegative_* die geeignete Clippingmethode ausgewählt.

[33] Tatsächlich verwenden beide sogar dieselbe Variable, damit keine neue Variable eingeführt werden muss, die denselben Nutzen erfüllt.

4.3 Darstellung multipler Volumen

Die Implementierung der Darstellung multipler Volumen wird in diesem Kapitel mit der Optimierung des gesamten Prozessornetzwerks zusammengefasst, da sie eng miteinander verwoben sind.

Bisher wird das Voreen Prozessornetzwerk bei der Initialisierung der Volume Rendering Komponente von MOPS 3D immer aus einer Datei gelesen und dann dynamisch angepasst. Dieses Dateiformat wird von Voreen selbst nicht mehr unterstützt, da es in VoreenVE durch das Projektdateiformat abgelöst wurde. Aus diesem Grund soll das Prozessornetzwerk komplett dynamisch erstellt werden, ohne etwas von Dateien einzulesen. Dazu werden bei der Erzeugung des Netzwerks alle benötigten Prozessoren erstellt und miteinander verknüpft. Ports werden dabei miteinander verbunden und Eigenschaften werden miteinander verknüpft.

Um die Performanz zu erhöhen, sollen keine Prozessoren unnötigerweise doppelt vorkommen, wie z.B. ein VolumeSource Prozessor für jedes Segment, obwohl die Segmente möglicherweise auf dasselbe Volumen zugreifen. Deshalb gibt es vom MultiCompositor und Canvas Prozessor jeweils nur eine Instanz und für jedes Volumen soll es nur einen VolumeSource Prozessor geben. Dies führt dazu, dass sich das Prozessornetzwerk in drei Teile untergliedern lässt: Der konstante Teil (MultiCompositor und Canvas), der Volumenteil (ein VolumeSource für jedes Volumen) und der Segmentteil (MeshProxyGeometry, MeshEntryExitPoints und SingleVolumeRaycaster). Abbildung 41 verdeutlicht diesen Aufbau des Prozessornetzwerks.

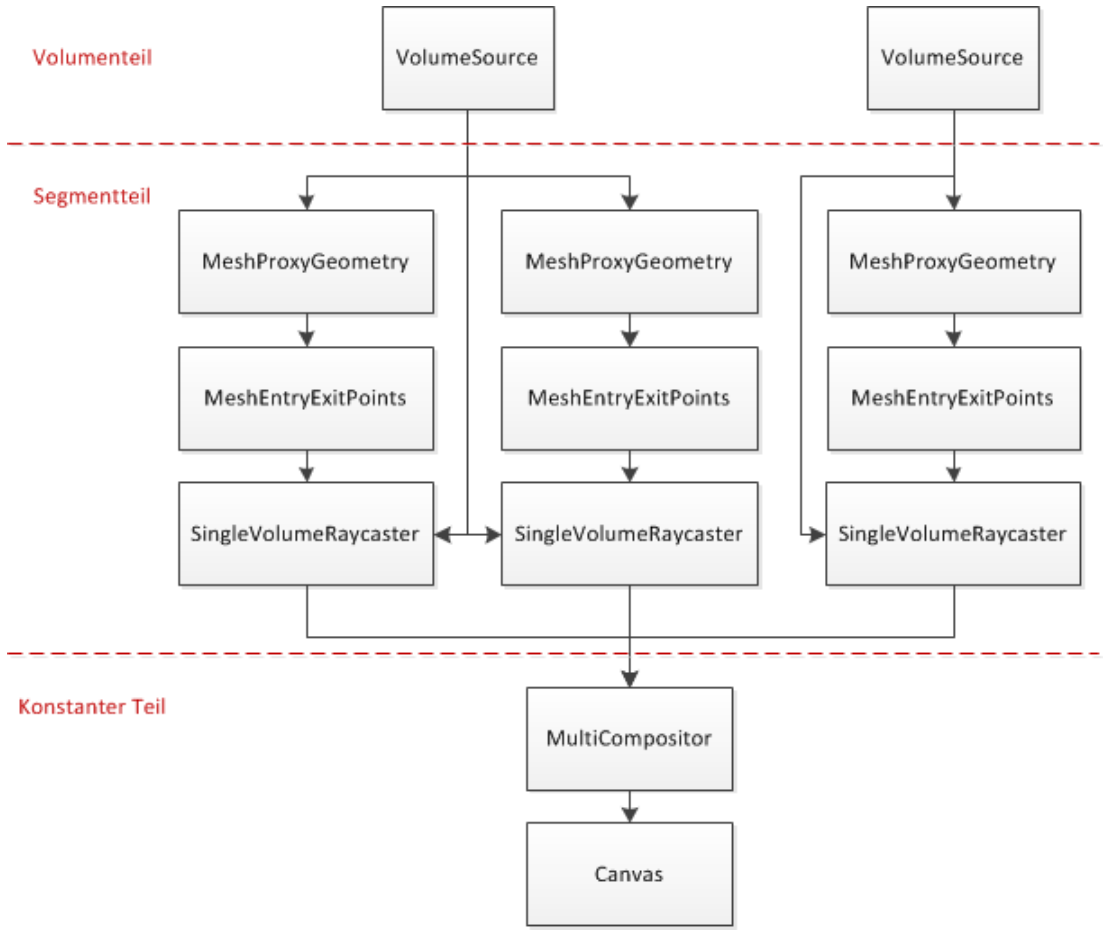


Abbildung 41: Optimierter Aufbau des Prozessornetzwerks mit multiplen Volumen

Quelle: eigene Darstellung

Um die Prozessoren der beiden variablen Teile voneinander unterscheiden zu können und einzeln ansprechbar zu machen, werden einzigartige Namen vergeben. Im Segmentteil setzt sich das aus dem Prozessornamen und einer dem Segment zugewiesenen Zahl zusammen. Im Volumenteil besteht dieser aus dem Prozessornamen und dem Namen des Volumens. Eine Besonderheit nimmt hier das neue Property Linking ein – das Verknüpfen von Eigenschaften (siehe Kapitel 4.1.1). Die Eigenschaften der verschiedenen Raycaster Prozessoren müssen immer identisch sein. Dazu gehören z.B. die Kamera, die Beleuchtungsparameter, die Abtastrate, etc. Um das Verknüpfen der Eigenschaften zu vereinfachen, wird der Raycaster eines neuen Segments immer mit den Raycaster-Eigenschaften des vorherigen Segments verknüpft. Beim Löschen eines Segments muss

darauf geachtet werden, dass der Nachfolger – falls es einen gibt – mit den Eigenschaften des Vorgängers des gelöschten Segments verknüpft wird und somit eine lückenlose Kette erhalten bleibt. Außerdem werden die Segmentzahlen der nachfolgenden Segmente angepasst, damit die Linearität der Zahlenreihe erhalten bleibt. Listing 11 stellt das Löschen eines Segments in Pseudocode dar.

```
Loesche alle beteiligten Prozessoren

Durchlaufe alle nachfolgenden Segmente
{
    Passe die Segmentzahl dieses Nachfolgers an

    Benenne die Prozessoren dieses Nachfolgers um

    Handelt es sich um den direkten Nachfolger
    {
        Verknuepfe seine Eigenschaften mit denen des direkten Vorgaengers
    }
}
```

Listing 11: Methode zum Löschen eines Segmentes

Für Volumen wird ähnlich vorgegangen. Bei der Initialisierung wird zunächst für jedes Volumen – wie auch beim Hinzufügen eines neuen Volumens – ein VolumeSource Prozessor erstellt und in das Prozessornetzwerk eingegliedert. Beim Löschen eines Volumens wird der zugehörige VolumeSource Prozessor anhand des Volumennamens ermittelt und mitsamt den zugehörigen Segmenten gelöscht. Wird ein Volumen umbenannt, so passt sich der Name des zugehörigen VolumeSource Prozessors automatisch an.

4.4 GPU-Segmentierung

Zur Implementierung der GPU-Segmentierung wird eine Klasse *GPUSegmentation*, deren Basisklasse die zuvor konstruierte Schnittstelle ist, in MOPS 3D integriert. Sowie eine Klasse *GPUSegmentationDlg*, die den zugehörigen Dialog repräsentiert und steuert. *GPUSegmentationDlg* stellt den Benutzeroberflächenteil dar, während *GPUSegmentation* als Teil der Anwendungslogik mit dem Framework zur GPU-Segmentierung kommu-

niziert. Das Framework selbst besteht aus einem eigenständigen Projekt, das – genau wie Voreen – in die MOPS 3D Projektmappe (Solution[34]) integriert ist und eine Abhängigkeit des MOPS 3D Projekts darstellt.

4.4.1 Initialisierung

Bei der Erstellung des Dialogs zur GPU-Segmentierung wird *GPUSegmentation* initialisiert. Dort wird ein Objekt der Klasse *SegmentationManager* erzeugt, welche Teil des GPU-Segmentierungsprojekts ist und als Grundgerüst für die Kommunikation mit Selbigem dient. *SegmentationManager* wurde von Marius Wirths im Rahmen seiner Diplomarbeit [Wir12] entworfen und implementiert jene Methoden der Schnittstelle, die auf Seiten der GPU-Segmentierung vorhanden sein müssen. Im Gegensatz dazu steht die *GPUSegmentation*, bei der die Methoden implementiert werden, die auf Seiten von MOPS 3D benötigt werden.
Sofern Volume Rendering nicht bereits aktiv ist, wird es an dieser Stelle ebenfalls initialisiert. Des Weiteren wird eine Kopie des Planungsdatensatzes erzeugt (d.h. eine Kopie des Volumens) und in das Voreen Prozessornetzwerk eingegliedert – unter dem Namen „GPUSegmentationTemp“. Alle Operationen der GPU-Segmentierung sollen auf dieser Kopie stattfinden, damit das Originalvolumen unverändert bleibt. Abbildung 42 verdeutlicht diesen Ablauf in einem Diagramm.

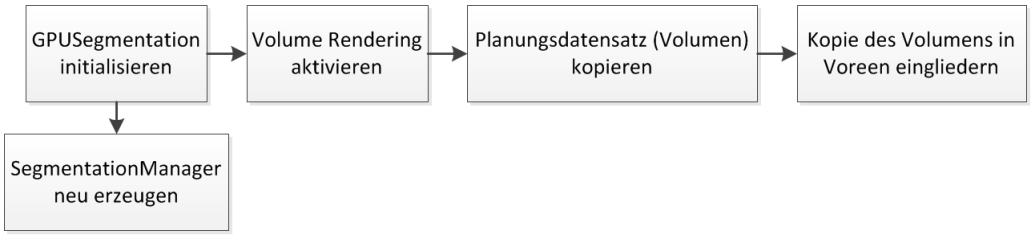


Abbildung 42: Ablauf der Initialisierung der GPU-Segmentierung

Quelle: eigene Darstellung

[34] In der Entwicklungsumgebung Microsoft Visual Studio lassen sich mehrere Projekte in einer Art Container zusammenfassen und in Abhängigkeit zueinander stellen. Diesen Container nennt man Solution oder Projektmappe.

4.4.2 Übergabe des Volumens

Anschließend fordert das Projekt der GPU-Segmentierung das Volumen, sowie dessen Größe an, um darauf operieren zu können. Da dieses von Voreen bereits als OpenGL 3D Textur im Grafikkartenspeicher gehalten wird, reicht es aus, die ID dieser Textur zu übergeben. Diese Texturen liegen Voreen üblicherweise im Format GL_ALPHA vor, um möglichst wenig Speicherplatz zu belegen. GL_ALPHA bedeutet, dass von jedem Voxel des Volumens nur der Alphawert gespeichert wird und somit nur ein Farbkanal benötigt wird. Dies ist ausreichend, da ohnehin nur die Intensität jedes Voxels bekannt ist und keine Farbinformationen vorliegen. Dieses Format führt jedoch zu Fehlern auf Seiten des GPU-Segmentierungsprojekts, da es von der darunterliegenden Programmierschnittstelle (OpenCL[35]) nicht unterstützt wird. Aus diesem Grund wird die Textur vor der Übergabe in das Format GL_RGBA (vier Farbkanäle) umgewandelt.

4.4.3 Saatpunkte und Zeichnungen

Das Einzeichnen von Saatpunkten und Nicht-Saatpunkten wird in die Mausereignisse[36] der Ausgabefenster integriert. Dort wird überprüft, ob der GPU-Segmentierungsdialog geöffnet ist und welcher Mausmodus (siehe Kapitel 3.3.1 unter „Saatpunkte“) ausgewählt ist. Handelt es sich dabei um den Modus für Freihandkonturen oder geschlossene Formen, so wird bei Drücken der Maustaste eine neue Zeichnung erstellt und ein erster Saatpunkt gesetzt. Eine Zeichnung beinhaltet alles, was zwischen dem Drücken und Loslassen der Maustaste gezeichnet wird. Sie wird durch die Klasse *Drawing* repräsentiert und dient als Container für Saatpunkte. Außerdem enthält sie die Art der Zeichnung. Diese hängt vom Mausmodus ab und kann entweder eine offene bzw. geschlossene Freihandzeichnung sein oder eine Linienzeichnung mit gefülltem Rechteck (bei großer Pinseldicke, siehe unten). Ein Saatpunkt wird durch die Struktur *Point* beschrieben. Sie beherbergt weiterhin die

[35] http://www.khronos.org/opencl

[36] MFC bietet sog. Ereignisse (Events) an, um Eingaben des Benutzers abzufangen. Sobald der Benutzer z.B. seine Maus bewegt wird das entsprechende Ereignis (MouseMove) ausgelöst und die zugehörige Methode wird aufgerufen.

Eigenschaft, ob es sich um einen Saatpunkt oder einen Nicht-Saatpunkt handelt. Abbildung 43 verdeutlicht das Verhältnis zwischen *Drawing* und *Point* in einem Diagramm. DrawingType ist dabei eine Variable vom Typ *enum*, welche die drei Arten von Zeichnungen enthält.

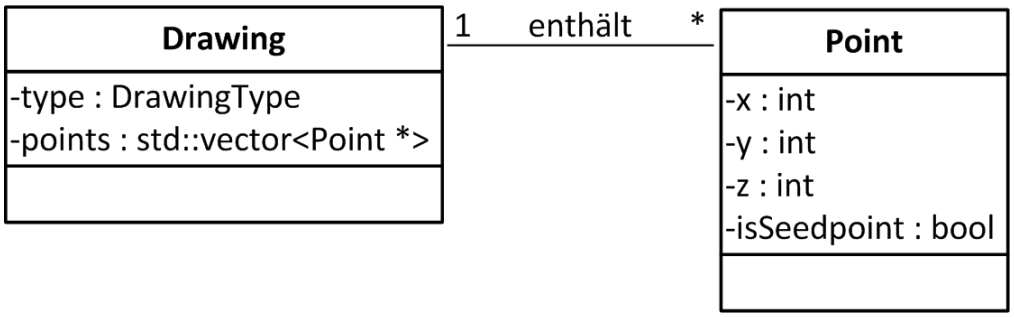


Abbildung 43: Klassendiagramm für Saatpunkt und Zeichnung

Quelle: eigene Darstellung

Beim Bewegen der Maus mit gedrückter Maustaste werden weiterhin Saatpunkte erstellt und beim Loslassen wird die Zeichnung beendet und mit all ihren erzeugten Punkten an das GPU-Segmentierungsprojekt übergeben. Die Art der erstellten Zeichnung hängt dabei von Mausmodus und Pinseldicke ab. Die Art der Saatpunkte dagegen hängt davon ab ob die STRG-Taste gedrückt ist. In dem Fall werden Nicht-Saatpunkte erstellt, ansonsten Saatpunkte. Abbildung 44 verdeutlicht den Ablauf der Mausinteraktionen in einem Diagramm.

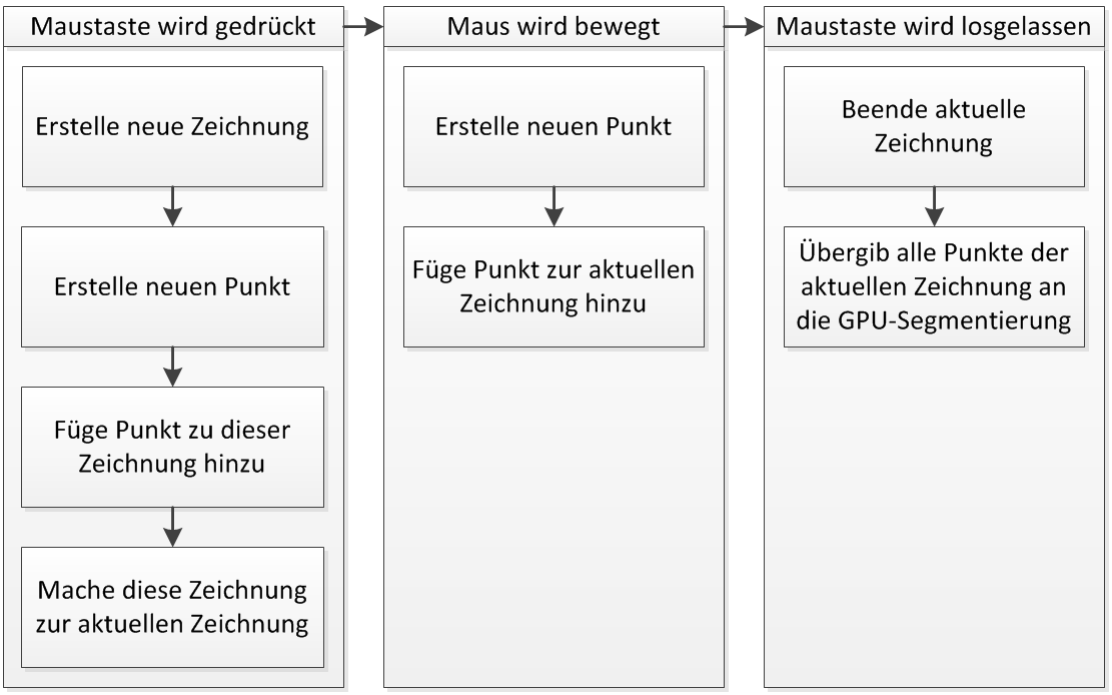


Abbildung 44: Ablauf der Erstellung von Zeichnungen und Saatpunkten im Zusammenhang mit der Mausinteraktion

Quelle: eigene Darstellung

Bei einer Pinseldicke größer als eins soll eine gerade Linie in die 2D Ansicht eingezeichnet werden, aus der ein Rechteck entsteht, das senkrecht zu der aktuellen Schicht steht (siehe Kapitel 3.3.1 unter „Saatpunkte“). Außerdem sollen die Kanten jeweils den gleichen Abstand zu der Schicht haben, in der sie in der 2D Ansicht eingezeichnet wurden. Beim Drücken der Maustaste wird dazu zunächst ermittelt, welche zwei Achsen in der 2D Ansicht dargestellt werden. Entlang der dritten, fehlenden Achse kann die entstehende Linie dann skaliert werden, um aus ihr ein Rechteck zu erstellen. Ausgehend von dem Punkt, der sich unmittelbar unter dem Mauszeiger befindet, werden daraufhin zwei Punkte erstellt, die sich auf dieser Achse befinden und beide den gleichen Abstand zum Ausganspunkt haben. Somit entspricht der Gesamtabstand zwischen den zwei Punkten der Pinseldicke. Beim Loslassen der Maustaste wird das Verfahren wiederholt, so dass ein zweidimensionales Rechteck entsteht, das durch die vier Punkte definiert wird. Listing 12 zeigt die Funktionsweise dieser Methode beispielhaft anhand der transversalen

Ansicht. Für die coronale und sagittale Ansicht ist die Vorgehensweise analog. Hierbei werden jeweils die y- und x-Koordinaten der Punkte verändert.

```
//Erstelle zwei Kopien des Ausgangspunkts
Point p1 = new Point(ausgangspunkt)
Point p2 = new Point(ausgangspunkt)

//Ermittle Soll-Abstand zwischen Kopie und Ausgangspunkt
abstand = (pinseldicke - 1) / 2;

//Veraendere die Koordinaten der Punkte abhaengig von der 2D Ansicht
//(Sagittal- und Coronalansicht funktionieren analog dazu)
if(ansicht == "transversal")
{
    p1.z += abstand;
    p2.z -= abstand;
}

//Fuege die Punkte der aktuellen Zeichnung hinzu
aktuelleZeichnung.add(p1);
aktuelleZeichnung.add(p2);
```

Listing 12: Methode zum Erstellen der Punkte des Pinseldicke-Rechtecks

Alle Zeichnungen werden mit OpenGL in allen Ansichten dargestellt. Geschlossene Zeichnungen werden dabei mit GL_LINE_LOOP gezeichnet, so dass Start- und Endpunkt automatisch verbunden werden. Freihandzeichnungen werden mit GL_LINE_STRIP gezeichnet, so dass alle Punkte miteinander verbunden sind (außer Start- und Endpunkt). Zeichnungen mit einer Pinseldicke größer als eins werden mit GL_QUADS als gefülltes Rechteck dargestellt. Von jeder Zeichnung werden stets nur die Punkte angezeigt, die sich auf derselben Schicht befinden wie bei der aktuellen 2D Ansicht. Damit sich die Zeichnungen auch immer im Vordergrund befinden und nicht vom Bilddatensatz verdeckt werden, werden sie leicht in Richtung Kamera versetzt. Das bedeutet in einer 2D Ansicht werden die Koordinaten jener Dimension verändert, die nicht dargestellt wird (bei der x-y-Ebene z.B. die z-Koordinate). Listing 13 verdeutlicht dieses Verfahren.

```
//Offset, der zu jeder Koordinate im glVertex3f() Aufruf addiert wird,
//um den Punkt im Vordergrund darzustellen
VGLVector3f offset(0.0f, 0.0f, 0.0f);
if(ref->displayMode == ViewManager::SAGITTAL)
    offset.x = getCurrentSlices()[0] + 0.2f;
else if(ref->displayMode == ViewManager::CORONAL)
    offset.y = getCurrentSlices()[1] + 0.2f;
else if(ref->displayMode == ViewManager::TRANSVERSAL)
    offset.z = getCurrentSlices()[2] + 0.2f;
```

Listing 13: Berechnung eines Offsets zum Zeichnen von Punkten im Vordergrund

Zusätzlich zu den Zeichnungen werden nach einer erfolgreichen GPU-Segmentierung alle zur Segmentierung gehörigen Punkte in den 2D Ansichten eingefärbt, um ihre Zugehörigkeit hervorzuheben.

4.4.4 Export/Import der GPU-Segmentierung

Um nach einer abgeschlossenen GPU-Segmentierung das Ergebnis in der herkömmlichen Segmentierung öffnen zu können, muss das Ergebnis zunächst von einer OpenGL 3D Textur in ein für MOPS 3D kompatibles VGL Volumen umgewandelt werden. Dazu wird über die Schnittstelle der GPU-Segmentierung eine Bytemaske des Volumens angefordert. In dieser Maske repräsentiert ein Byte jeweils ein Voxel und zeigt an, ob dieser Voxel zum Segmentierungsergebnis gehört (d.h. der Wert ist größer als 0) oder nicht. Anhand dieser Maske lässt sich ein VGL Volumen erstellen, das man der Anwendungslogik der herkömmlichen Segmentierung nach dem Öffnen ihres Dialogs problemlos zuweisen kann. Listing 14 verdeutlicht das Verfahren der Umwandlung.

```
//Neues Volumen erstellen
VGLSampleGrid *volume = new VGLSampleGridData(VGL_TYPE_INT16,
    VGLSampleGridSize(volumeSize[0], volumeSize[1], volumeSize[2]));

void *ptrNew, *ptrSeg;
GLubyte *mask = getVolumeMask(); //Bytemaske anfordern

for (unsigned int z=0; z<volumeSize[2]; z++)
{
    for(unsigned int y=0; y<volumeSize[1]; y++)
    {
        for(unsigned int x=0; x<volumeSize[0]; x++)
        {
            //Pointer auf das Voxel im VGL Volumen
            ptrNew = volume->getVoidPtr(x,y,z);

            //Pointer auf das Voxel in der Bytemaske
            ptrSeg = (void *)(mask + z * volumeSize[0] * volumeSize[1] +
                y * volumeSize[0] + x);

            //Voxel gehoert zum Segmentierungsergebnis
            if(*(GLubyte *)ptrSeg > 0)
                //Setze die Intensitaet im VGL Volumen auf den Maximalwert
                *(VGLuint16 *)ptrNew = 65535;
            else
                *(VGLuint16 *)ptrNew = 0;
        }
    }
}
```

Listing 14: Umwandlung eines GPU-Segmentierungsergebnisses in ein VGL Volumen

Da die Bytemaske ein Pointer ist, können die einzelnen Bytes nicht über die Koordinaten angesprochen werden, wie z.B. beim VGL Volumen (*volume->getVoidPtr(x,y,z)*). Stattdessen muss auf sie über ein Offset zugegriffen werden, ähnlich wie der Index bei einem Array. Um eine Koordinate in einen Index umzuwandeln, muss die Größe des Volumens bekannt sein. Daraus lässt sich der Index nach folgender Formel berechnen:

$$Index(x, y, z) = z \cdot dim_x \cdot dim_y + y \cdot dim_x + x \tag{4.3}$$

dim_x und dim_y sind dabei Breite und Höhe des Volumens.
Das Importieren einer Segmentierung aus der herkömmlichen Segmentierung in die GPU-Segmentierung funktioniert genau umgekehrt. Das VGL Volumen wird nach dem glei-

chen Prinzip wie oben in eine Bytemaske umgewandelt und dann über die Schnittstelle an die GPU-Segmentierung übergeben, damit diese das Segmentierungsergebnis wieder als OpenGL 3D Textur darstellen kann.

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Arbeit präsentiert. Dazu wird zum einen die Performanz vor und nach den implementierten Änderungen gemessen und miteinander verglichen. Zum anderen werden Anwendungsbeispiele zu den funktionalen Erweiterungen beschrieben und der entsprechende Interaktionsfluss des Benutzers vorgestellt. Anschließend wird die GPU-Segmentierung separat präsentiert, da sie eine eigenständige Neuerung darstellt. Alle Szenarien werden auf der Testmaschine (siehe Kapitel 3.1) durchgeführt und verwenden verschiedene Bilddatensätze[37], um die Ergebnisse nicht zu verfälschen.

5.1 Performanz

Um den Vergleich der Performanzunterschiede nachvollziehen zu können, muss vorher die Änderung vorgestellt werden, die den Qualitätsfaktor (siehe Kapitel 4.1.3) betrifft. Im globalen Volume Rendering Konfigurationsdialog lässt sich die Samplingrate nun direkt über einen Schieberegler einstellen, anstatt auf sechs Voreinstellungen angewiesen zu sein. Abbildung 45 zeigt den Dialog mit dem neuen Schieberegler.

[37] Es werden Daten von verschiedenen Patienten verwendet und von verschiedenen bildgebenden Verfahren (CT, MRT).

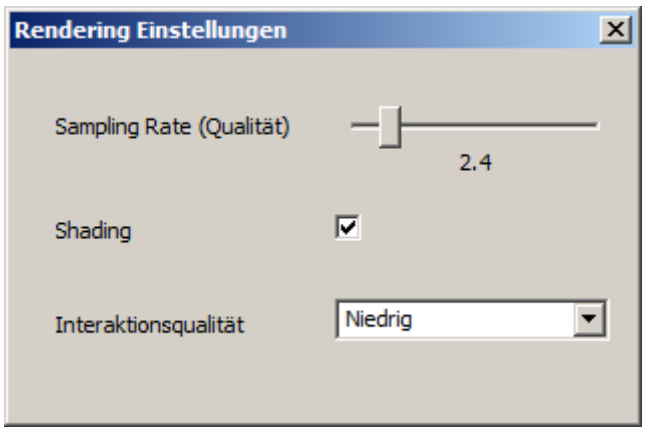


Abbildung 45: Globaler Volume Rendering Konfigurationsdialog mit Schieberegler für die Samplingrate

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Die Auswirkungen der Änderungen auf die Performanz werden mit Hilfe des Programms FRAPS[38] anhand der Darstellungsgeschwindigkeit in Bildern pro Sekunde gemessen. Da es bei der Darstellung nur während der Interaktion (bzw. nur dann, wenn eine Aktualisierung der Anzeige nötig ist) auf die Geschwindigkeit ankommt, wird die Durchschnittsgeschwindigkeit bei einer zehnsekündigen Rotation der 3D Ansicht ermittelt. Dieses Ergebnis wird mit der Geschwindigkeit verglichen, die das Programm vor Implementierung der Änderungen erreichte. Damit ein direkter Vergleich möglich ist, muss Volume Rendering bei beiden Versionen identisch konfiguriert sein. Da in der neueren Version der Qualitätsfaktor durch die Samplingrate ersetzt wurde (siehe Kapitel 4.1.3), muss eine Umrechnung erfolgen, um die Werte anzugleichen. Beide Werte dienen der Abtastrate für das Raycasting. Bei der alten Voreen Version errechnet sich die Schrittweite beim Wachsen des Strahls wie folgt:

$$t_{Incr} = 0,005 \cdot \frac{1}{f(q)} \tag{5.1}$$

t_{Incr} ist dabei die Schrittweite und $f(q)$ ist als unstetige Funktion zu betrachten, die nur

[38] http://www.fraps.com

an diskreten Stellen Werte besitzt:

$$f(q) = \begin{cases} 0,2 & \text{für } q = \text{„very low“} \\ 0,5 & \text{für } q = \text{„low“} \\ 1 & \text{für } q = \text{„normal“} \\ 2 & \text{für } q = \text{„high“} \\ 5 & \text{für } q = \text{„higher“} \\ 10 & \text{für } q = \text{„highest“} \end{cases} \tag{5.2}$$

q ist eine der sechs vorgegebenen Qualitätseinstellungen. Somit nimmt t_{Incr} Werte zwischen 0,025 und 0,0005 an.

Für die neue Voreen Version berechnet sich die Schrittweite folgendermaßen:

$$t_{Incr} = \frac{1}{dim_{max} \cdot s_r} \tag{5.3}$$

dim_{max} ist hierbei der Maximalwert von Breite, Höhe und Tiefe des Volumens und s_r ist die Samplingrate, die durch den Benutzer regulierbar ist. Sie besitzt einen Wert zwischen 0,01 und 20. Demnach liegt t_{Incr} bei einer Standardvolumengröße von 256 x 256 x 256 zwischen 0,39 und 0,0002.

Da die Schrittweite bei der alten Version nur durch sechs Voreinstellungen beeinflusst werden kann, macht es Sinn, die Schrittweite der neuen Version der alten anzugleichen für die Messung. Stellt man die beiden Formeln für t_{Incr} gegenüber und löst sie nach s_r auf, so erhält man eine Formel für die Samplingrate (in Abhängigkeit von q), mit der jeweils dieselbe Schrittweite wie in der alten Version herauskommt (für dim_{max} wird wieder der Wert 256 angenommen):

$$s_r(q) = \begin{cases} 0,16 & \text{für } q = \text{„very low“} \\ 0,40 & \text{für } q = \text{„low“} \\ 0,78 & \text{für } q = \text{„normal“} \\ 1,56 & \text{für } q = \text{„high“} \\ 3,90 & \text{für } q = \text{„higher“} \\ 7,81 & \text{für } q = \text{„highest“} \end{cases} \tag{5.4}$$

Anhand dieser Umrechnung lässt sich die Performanz beider Versionen vergleichen. Dazu wird bei den Messungen derselbe Datensatz mit den gleichen Einstellungen zu Volume Rendering und identischer Transferfunktion verwendet. Es wird jeweils eine Messreihe für alle Qualitätsstufen durchgeführt. Um auch den Einfluss der Interaktionsqualität (Interaction Coarseness, siehe Kapitel 2.3) beobachten zu können, werden zudem Messungen für alle Interaktionsqualitätseinstellungen vorgenommen. Hiervon gibt es drei verschiedene Stufen, die jeweils einen numerischen Wert repräsentieren, der die Intensität der Bildverschlechterung[39] angibt: *niedrig* (4), *normal* (2) und *hoch* (1). Das Besondere an der Interaktionsqualitätsstufe *hoch* ist, dass ihr Wert von 1 bedeutet, dass das Bild gar nicht verschlechtert wird, d.h. sie steht für maximale Interaktionsqualität. Zusätzlich werden alle Messungen mit und ohne Clipping durchgeführt (mit einem festgelegten Quader als Clippingstruktur). Die Ergebnisse werden in Abbildung 46 dargestellt.

[39] Die Interaktionsqualität (Interaction Coarseness) verschlechtert das Bild während der Interaktion zugunsten der Darstellungsgeschwindigkeit.

Qualität	Version	Bilder pro Sekunde					
		ohne Clipping			mit Clipping		
		niedrig*	normal*	hoch*	niedrig*	normal*	hoch*
very low (s_r = 0,2)	alt	50	48	45	40	40	39
	neu	58	57	64	57	56	59
low (s_r = 0,4)	alt	45	42	37	40	40	39
	neu	58	57	60	56	56	59
normal (s_r = 0,8)	alt	40	35	28	40	40	39
	neu	54	48	41	56	56	59
high (s_r = 1,6)	alt	31	26	20	40	39	37
	neu	41	33	24	56	56	59
higher (s_r = 4,0)	alt	19	15	10	38	37	35
	neu	23	17	12	56	56	59
highest (s_r = 7,8)	alt	12	9	6	36	33	31
	neu	13	10	7	55	55	59

* Interaktionsqualität (Interaction Coarseness)

Abbildung 46: Ergebnisse der Performanzmessungen für die alte und neue Version von Voreen in MOPS 3D

Quelle: eigene Darstellung

Alle Angaben sind in Bildern pro Sekunde. Die Qualitätsstufen (und ihre entsprechenden Samplingrates s_r) *very low* und *low* führen zu einer Darstellungsqualität, die in der Praxis unbrauchbar ist. Auf der anderen Seite gibt es zwischen den Stufen *higher* und *highest* keine erkennbare Verbesserung (entsprechende Zeilen sind in der Tabelle blau eingefärbt).

Wie sich zeigt, ist ein durchgängiger Geschwindigkeitszuwachs zu erkennen, der ohne Clipping leicht ausfällt und mit Clipping groß ist. Dieser Zuwachs ist hauptsächlich der Integration der neuen Voreen Version zuzuschreiben, die einige Performanzverbesserungen mit sich bringt und nicht mehr auf OpenGL Version 2.1 beschränkt ist. Ein Trend, der sich bei den Messwerten der neuen Version beobachten lässt, ist, dass die Anwendung bei einer Interaktionsqualität von *hoch* oft schneller läuft als mit niedrigerer Interakti-

onsqualität (bei gleicher Abtastrate), obwohl bei steigender Interaktionsqualität eine Abnahme der Performanz zu erwarten ist (in Abbildung 46 sind die betroffenen Tabelleneinträge grün eingefärbt). Dies ist dadurch zu erklären, dass der Rechenaufwand der Interaction Coarseness ihren Nutzen teilweise (bei niedriger Darstellungsqualität s_r oder aktivem Clipping) übersteigt, d.h. ihre Berechnung kostet mehr Performanz, als die Bildverschlechterung liefert. Die Interaktionsqualität *hoch* kommt einem Rechenaufwand von Null gleich, da Interaction Coarseness in diesem Fall gar nicht angewandt wird (siehe oben).

Für eine weitere Performanzverbesserung sorgt die neu implementierte Deaktivierung von Segmenten (siehe Kapitel 3.1.2). Ist ein Segment deaktiviert, so stellt die 3D Ansicht so viele Bilder pro Sekunde dar, als existiere das Segment gar nicht. Das bedeutet, bei einem aktiven und einem inaktiven Segment ist die Darstellungsgeschwindigkeit genau so hoch wie bei nur einem Segment. Und bei nur einem inaktiven Segment ist sie so hoch als wäre Volume Rendering global deaktiviert. Dies wird in einer weiteren Messreihe verdeutlicht. Dazu werden für beide Versionen Messungen durchgeführt mit jeweils zwei aktiven Segmenten, einem aktiven und einem inaktivem Segment, sowie mit zwei inaktiven Segmenten. Dabei haben beide Segmente eine identische Transferfunktion und verwenden kein Clipping. Die Qualitätsstufen *very low*, *low* und *highest* werden bei dieser Messung nicht berücksichtigt. Abbildung 47 stellt die Ergebnisse dieser Messungen in einer Tabelle dar.

Qualität	Version	Bilder pro Sekunde								
		2 aktive Segmente			1 aktives, 1 inaktives Segment			2 inaktive Segmente		
		niedrig*	normal*	hoch*	niedrig*	normal*	hoch*	niedrig*	normal*	hoch*
normal (s_r = 0,8)	alt	30	25	19	38	34	28	39	39	38
	neu	36	29	26	49	45	43	65	65	65
high (s_r = 1,6)	alt	22	17	13	30	25	20	39	39	38
	neu	25	19	15	40	33	25	65	65	65
higher (s_r = 4,0)	alt	12	9	6	19	15	11	39	39	38
	neu	13	9	7	22	17	12	65	65	65

* Interaktionsqualität (Interaction Coarseness)

Abbildung 47: Ergebnisse der Performanzmessungen für mehrere Segmente

Quelle: eigene Darstellung

Hierbei ist der große Geschwindigkeitsunterschied für zwei inaktive Segmente zu erkennen. Bei einem aktiven und einem inaktiven Segment fällt er allerdings nicht so groß aus.

5.2 Funktionale Erweiterung

5.2.1 Form der Clippingstruktur

Das Ellipsoid ist als neu eingeführte Clippingstruktur in den Segment-Editor eingebunden. Um diesen auszuwählen, muss der Benutzer ihn im Dropdown-Feld „Clip Geometrie“ anwählen, woraufhin das Clipping automatisch umgeschaltet wird. Abbildung 48 zeigt den neuen Segment-Editor mit dem Feld für die Clippingstruktur.

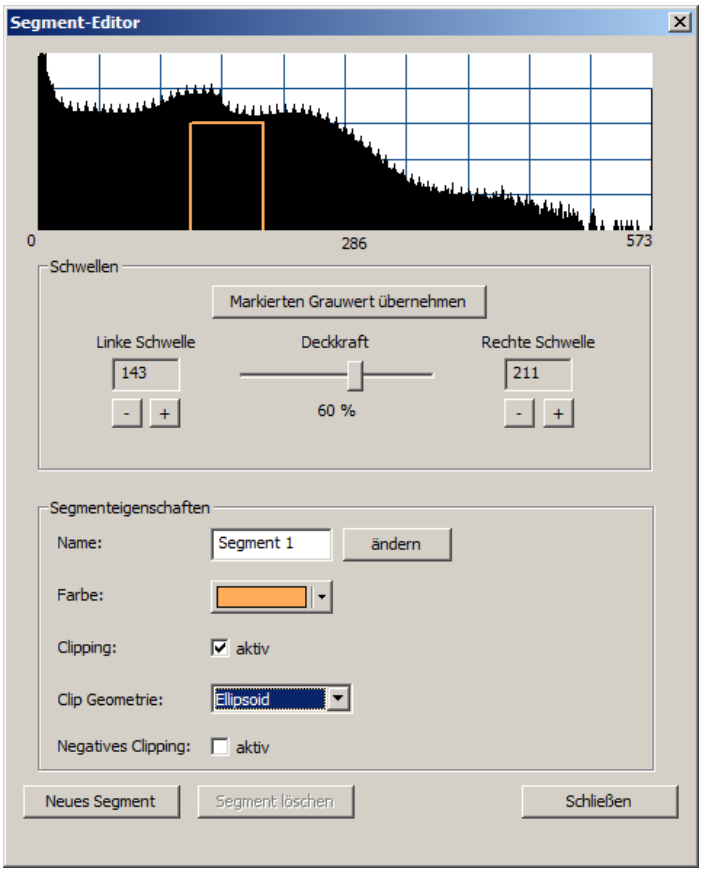


Abbildung 48: Segment-Editor mit Feldern zur Clippingstruktur und inversem Clipping
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Das Skalieren, Verschieben und erneute Aufziehen eines Ellipsoids funktioniert – genau wie beim Quader – mit der Maus, mit dem Unterschied, dass in den 2D Ansichten eine Ellipse statt eines Rechtecks gezeichnet wird. Abbildung 49 verdeutlicht diesen Sachverhalt anhand eines Tumors, der durch Clipping mit Hilfe eines Ellipsoids begrenzt wird.

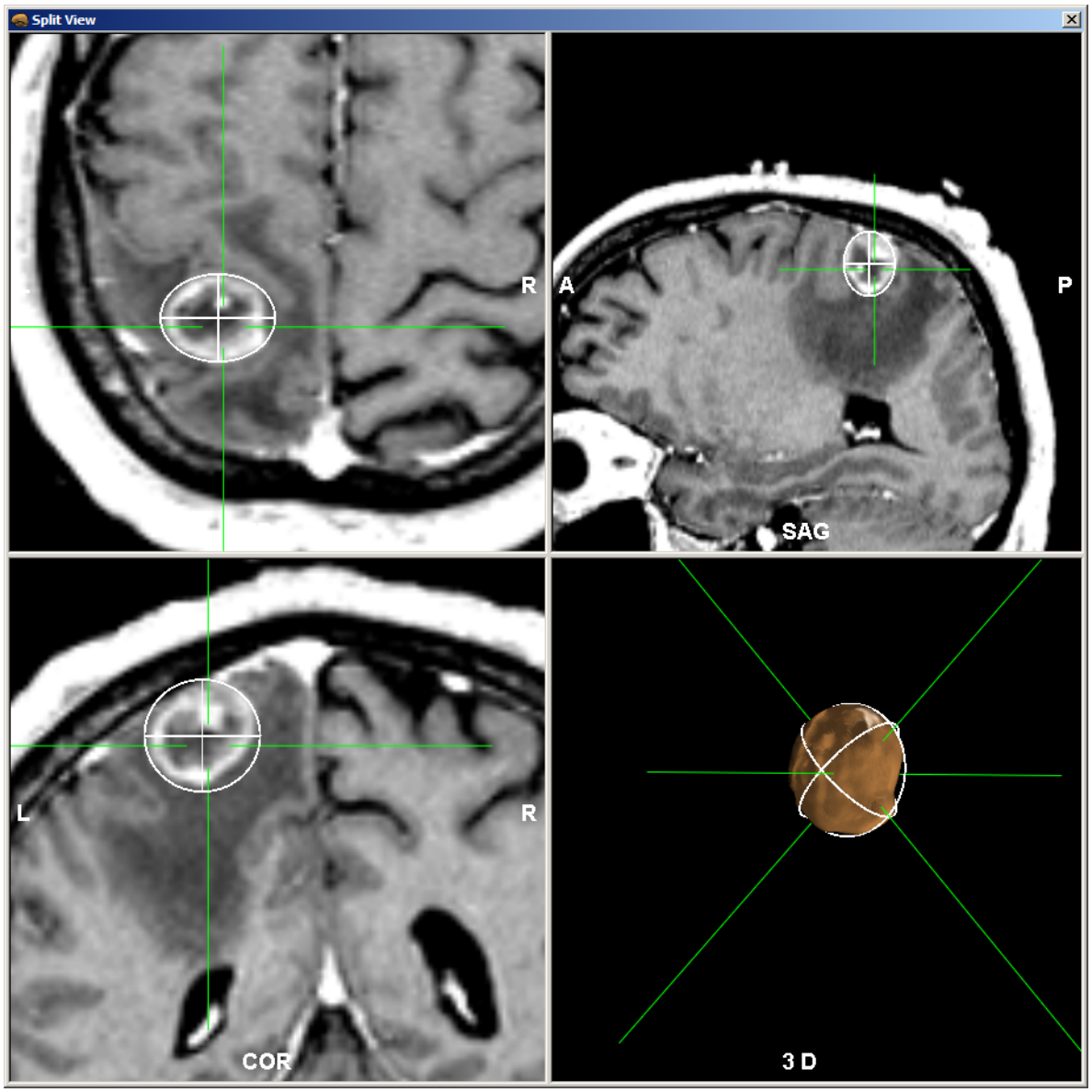
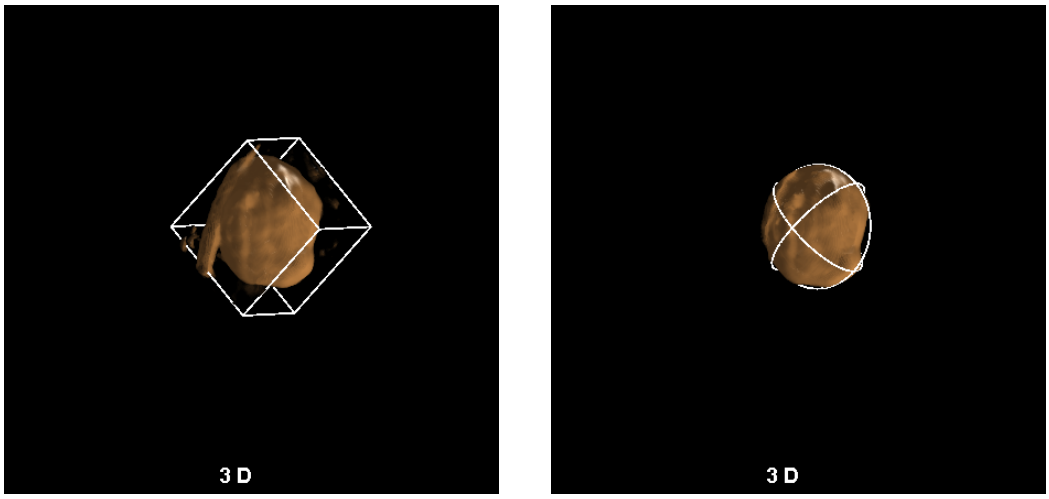


Abbildung 49: Clipping eines rundlichen Tumors durch ein Ellipsoid

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Hierbei ist deutlich zu erkennen, dass sich dieser rundliche Tumor mit einer gut gewählten Transferfunktion sauber eingrenzen lässt – besser als das mit einem Quader als Clippinstruktur möglich wäre. Zum Beweis stellt Abbildung 50 Clipping für Quader und Ellipsoide im direkten Vergleich am selben Tumor mit unveränderten Clippinggrenzen vor.



(a) Quader (b) Ellipsoid

Abbildung 50: Tumor mit Quader und Ellipsoid als Clippingstruktur
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Clipping für Ellipsoide läuft mit steigender Darstellungsqualität deutlich langsamer als für Quader. Das liegt daran, dass bei einer höheren Darstellungsqualität mehr Voxel abgetastet werden und der Fragment Shader somit auch öfters ausgeführt wird. Für die Qualitätseinstellungen *normal* und *high* ist die Performanz aber ausreichend, um vom menschlichen Auge als durchgehend flüssig wahrgenommen zu werden. Abbildung 51 verdeutlicht dies anhand einer Tabelle mit Messergebnissen der Performanz beider Clippingverfahren.

Qualität	Clippingstruktur	Bilder pro Sekunde		
		niedrig*	normal*	hoch*
normal (s_r = 0,8)	Quader	56	56	59
	Ellipsoid	49	42	33
high (s_r = 1,6)	Quader	56	56	59
	Ellipsoid	36	27	19
higher (s_r = 4,0)	Quader	56	56	59
	Ellipsoid	19	13	9

* Interaktionsqualität (Interaction Coarseness)

Abbildung 51: Ergebnisse der Performanzmessungen für Clipping mit Quader und Ellipsoid

Quelle: eigene Darstellung

5.2.2 Inverses Clipping

Der Benutzer kann das Clipping invertieren, indem er die Checkbox „Negatives Clipping“ im Segment-Editor (siehe Abbildung 48) aktiviert. Unmittelbar nach dem Aktivieren wird das Clipping dann automatisch invertiert. Abbildung 52 zeigt die Volume Rendering Darstellung eines Kopfes mit inversem Clipping für Quader. Die Transversalebene der MR-Schichten wird als einzige in der 3D Ansicht dargestellt, um zu verdeutlichen, dass sie einen guten Einblick auf die Ventrikel bietet.

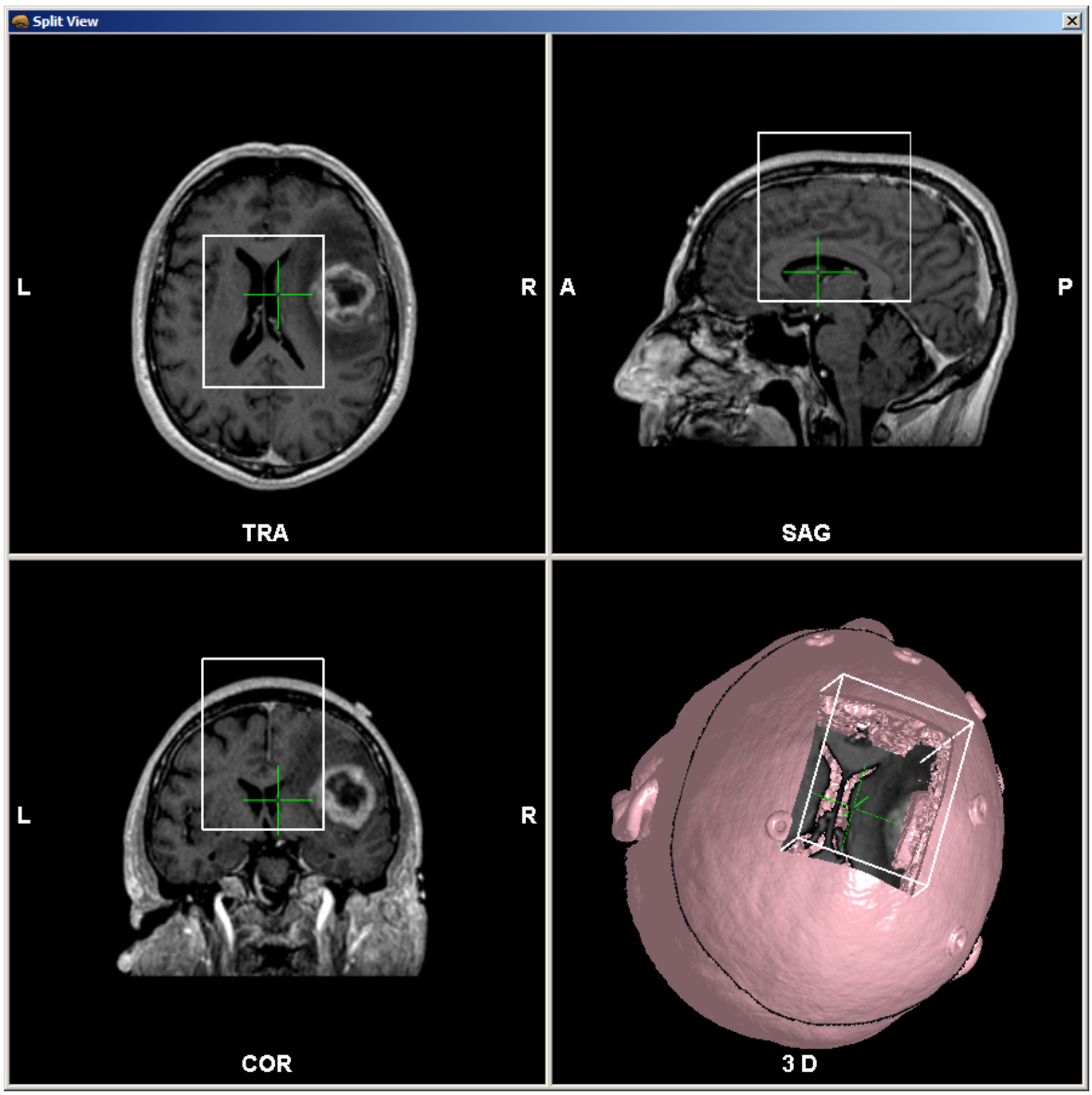


Abbildung 52: Kopf mit Einblick auf die Ventrikel, mit inversem Clipping für Quader
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Zum Vergleich dazu zeigt Abbildung 53 den selben Datensatz mit Clipping für Ellipsoide (die Clippinggrenzen wurden verändert).

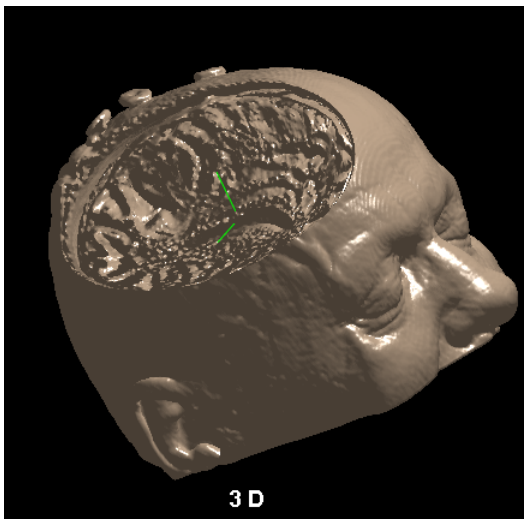


Abbildung 53: Kopf mit inversem Clipping für Ellipsoide
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Inverses Clipping läuft – unabhängig von der Struktur – vergleichbar schnell zu normalem Clipping für Ellipsoide, da beide das gleiche Verfahren benutzen.

5.2.3 Multiple Volumen

Das Verfahren zum Hinzufügen weiterer Volumen (neben dem Planungsdatensatz) wird für die GPU-Segmentierung benötigt (siehe Kapitel 3.2.3 und Kapitel 4.4). Dort wird während der Initialisierung eine Kopie des Planungsdatensatzes als neues Volumen automatisch hinzugefügt. Der Benutzer ist nicht imstande manuell ein neues Volumen hinzuzufügen. Das Löschen von vorhandenen Volumen ist ihm jedoch jederzeit zugänglich – sofern es sich nicht um den Planungsdatensatz handelt. Hierzu muss er das zu löschende Volumen im Strukturbrowser anklicken, um den Dialog zum Hinzufügen eines neuen Segments zu öffnen. Dort kann er bei löschbaren Volumen den Button „Datensatz löschen“ betätigen, um das Volumen zu entfernen. Abbildung 54 zeigt den Strukturbrowser mit zwei Volumen.

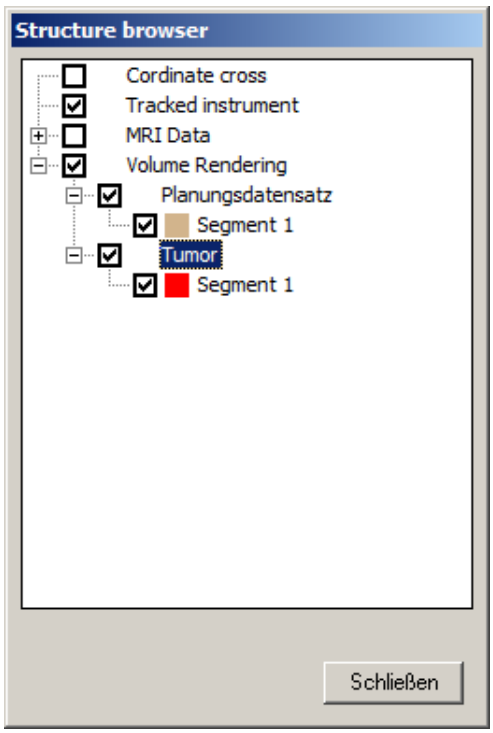


Abbildung 54: Strukturbrowser mit zwei Volumen

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

5.3 GPU-Segmentierung

In diesem Kapitel wird die Benutzeroberfläche zur Steuerung der GPU-Segmentierung vorgestellt. Dazu wird beispielhaft etwas per Region Growing segmentiert. Dieser Algorithmus und die gesamte inhaltliche Implementierung der GPU-Segmentierung wurde von Marius Wirths in seiner Diplomarbeit [Wir12] entwickelt. Hierbei steht jedoch stets die Benutzeroberfläche im Vordergrund, dessen Entwicklung Teil dieser Arbeit war, während die Verwendung der tatsächlichen GPU-Segmentierung nur zu Vorstellungszwecken dient.

Die GPU-Segmentierung lässt sich genau wie die herkömmliche Segmentierung über die Menüleiste starten. Der Dialog, der sich daraufhin öffnet, wird in Abbildung 55 dargestellt.

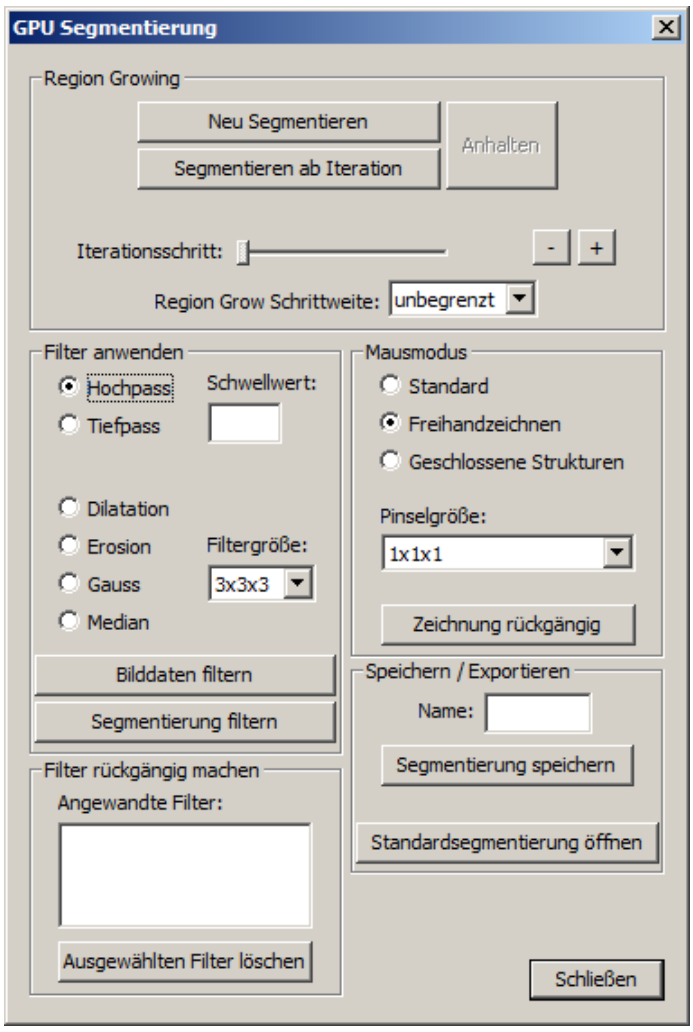
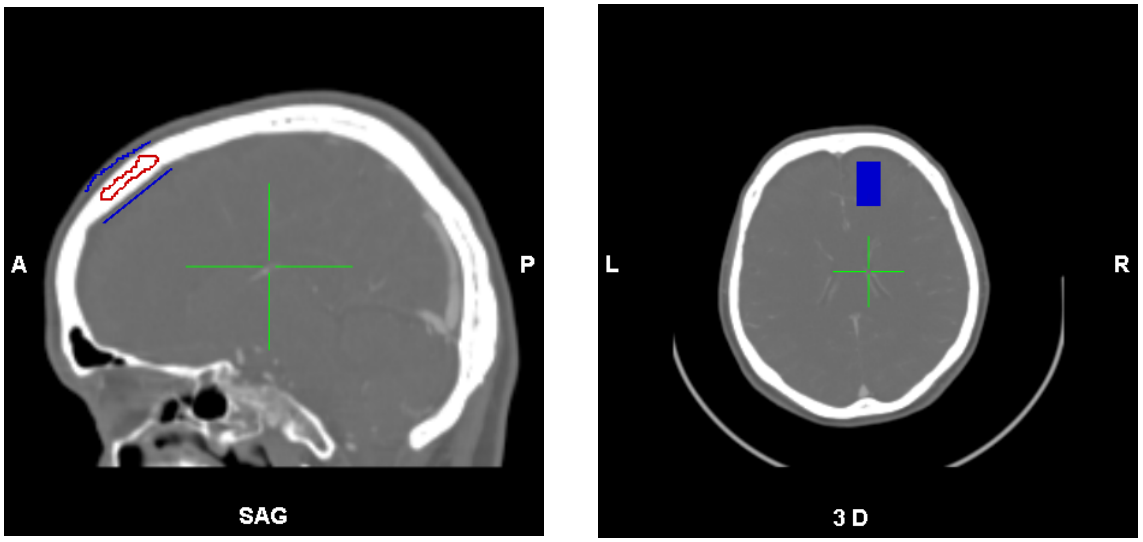


Abbildung 55: Dialog zur GPU-Segmentierung im Ausgangszustand

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Zuallererst muss der Benutzer mindestens einen Saatpunkt festlegen. Meist ist es jedoch sinnvoll, mehrere Saatpunkte (und Nicht-Saatpunkte) zu definieren. Im Dialog ist dafür die Gruppierung „Mausmodus“ zuständig. Man wählt den gewünschten Modus aus, stellt ggf. die Pinseldicke ein und zeichnet direkt in die 2D Ansicht (mit gedrückter STRG-Taste für Nicht-Saatpunkte). Der Button „Zeichnung rückgängig“ löscht die jeweils zuletzt erstellte Zeichnung. Abbildung 56 zeigt beispielhaft Zeichnungen zur Segmentierung der Schädelknochen eines CT-Datensatzes. Die Zeichnungen sind (von oben nach unten im linken Bild): Freihandzeichnung mit gedrückter STRG-Taste, geschlossene

Freihandzeichnung, Freihandzeichnung mit gedrückter STRG-Taste bei einer Pinseldicke von 1x1x15. Im rechten Bild ist deutlich das Rechteck zu erkennen, dass durch die Zeichnung mit hoher Pinseldicke ensteht.



(a) 2D Ansicht

(b) 3D Ansicht von oben (transversal)

Abbildung 56: Saatpunktzeichnungen (Saatpunkte sind rot, Nicht-Saatpunkte sind blau)
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

In der Gruppierung „Region Growing“ findet man anschließend den Button „Neu Segmentieren“, der den Region Growing Algorithmus startet. Während der Prozess läuft, kann man die Region in der 3D Ansicht beim Wachsen beobachten. Auch Kameraaktionen wie Rotation sind währenddessen möglich, da die GPU-Segmentierung parallel zum Hauptprogramm stattfindet. Nachdem das Region Growing beendet ist, oder durch den „Anhalten“ Button unterbrochen wurde, sieht man das Segmentierungsergebnis sowohl in der 3D Ansicht als auch in den 2D Ansichten. Im Letzteren wird es durch grün eingefärbte Pixel dargestellt. Abbildung 57 zeigt eine solche Segmentierung.

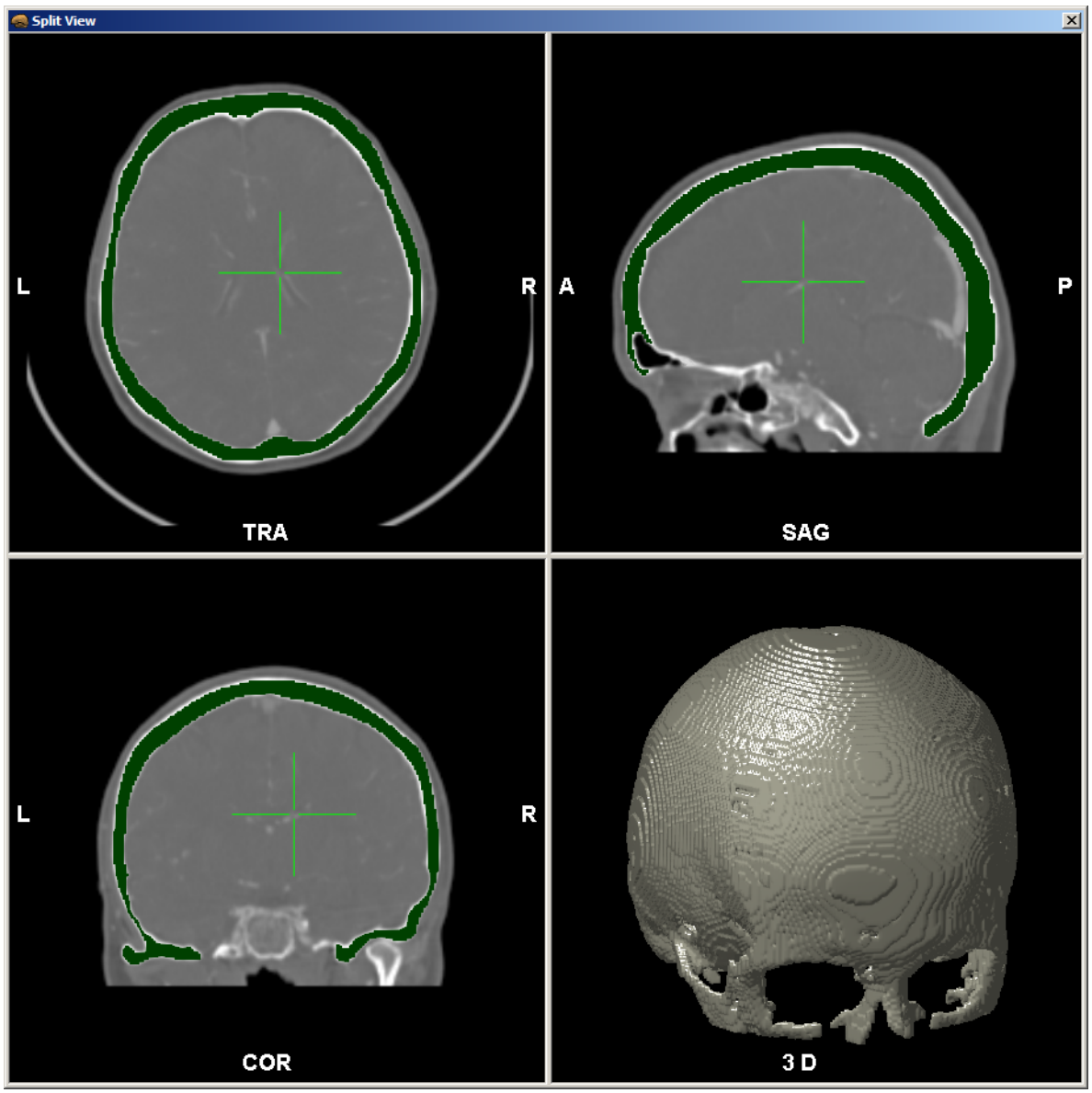
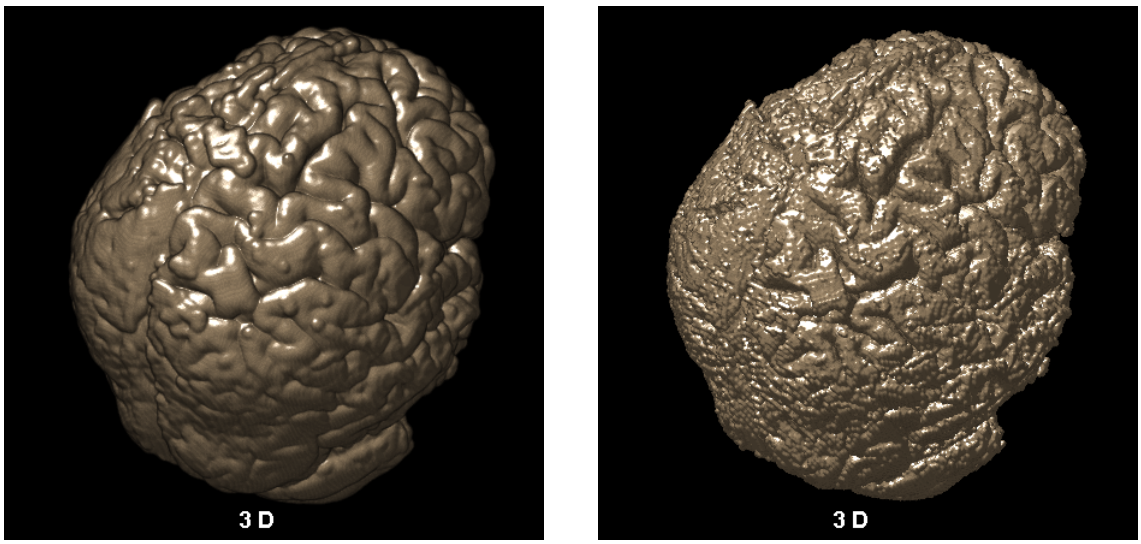


Abbildung 57: GPU-Segmentierungsergebnis

Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Es ist deutlich zu erkennen, dass die 3D Ansicht sehr grobkörnig wirkt, da die Grenzen der einzelnen Voxelschichten gut zu sehen sind. Der Grund dafür wird in der Arbeit von Marius Wirths (siehe [Wir12] Kapitel 5.1.1) ausführlich behandelt. Dort wird auch beschrieben, dass sich dieses Phänomen durch geeignete Filterung des Segmentierungsergebnisses mildern lässt und somit eine bessere Darstellungsqualität erreicht wird. Die

Filter befinden sich im Dialog unter der Gruppierung „Filter anwenden“. Der Benutzer wählt dort einen der sechs verschiedenen Filter aus und passt den jeweiligen Parameter an. Im Falle von Hoch- und Tiefpassfilter ist das der Schwellwert, der nicht über- bzw. unterschritten werden soll. Bei den anderen vier Filtern ist das die Anzahl der benachbarten Voxel, die mit in die Berechnung einbezogen werden bzw. kurzum die Filtergröße. Anschließend führt ein Klick auf einen der Buttons „Bilddaten filtern“ oder „Segmentierung filtern“ den Filter aus. Der erstgenannte Button führt den Filter auf den Voxeln des Eingangsvolumens aus (d.h. vor der Segmentierung) um den Region Growing Prozess zu begünstigen. Folglich muss nach seiner Anwendung (erneut) segmentiert werden, um ein Ergebnis zu erhalten. Der zweite Button führt den Filter direkt auf den Voxeln des Segmentierungsergebnisses aus und dient speziell dazu, die Darstellungsqualität zu erhöhen. Abbildung 58 zeigt beispielhaft die Darstellungsqualität einer Segmentierung, die mit einem Gaussfilter der Größe 5x5x5 verbessert wurde, im Vergleich mit derselben Segmentierung ohne Filterung.



(a) Gaussfilter (5x5x5) (b) keine Filterung

Abbildung 58: Vergleich eines Segmentierungsergebnisses mit und ohne Filterung
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

Nachdem ein Filter hinzugefügt wurde, taucht dieser in der Liste „Angewandte Filter“

unter der Gruppierung „Filter rückgängig machen“ auf. Dort kann der Benutzer einen beliebigen Filter auswählen und ihn mit einem Klick auf „Ausgewählten Filter löschen“ wieder entfernen und dessen Anwendung rückgängig machen.
Ist das Region Growing einmal durchgelaufen (oder vorzeitig angehalten worden), kann man mit dem Schieberegler „Iterationsschritt“ in der Gruppierung „Region Growing“ die einzelnen Iterationen des Region Growing durchblättern. Die 3D Ansicht aktualisiert sich dabei automatisch, so dass man die einzelnen Iterationen in Echtzeit sieht. Mit dem Button „Segmentieren ab Iteration“ startet der Segmentierungsalgorithmus nicht komplett neu, sondern von der aktuell angewählten Iteration aus. Auf diese Weise lassen sich Segmentierungen schnell korrigieren, z.B. wenn eine Segmentierung gegen Ende des Region Growing Algorithmus über eine Grenze hinausläuft, die sie nicht überschreiten soll. In diesem Fall wählt man die Iteration an, in der die Region ausläuft, fügt eine begrenzende Zeichnung hinzu und segmentiert ab dieser Iteration neu. Des Weiteren kann der Benutzer unter „Region Grow Schrittweite“ vor einer Segmentierung spezifizieren, wie schnell die Segmentierung in einer Iteration wachsen soll, d.h. wie viele benachbarte Voxel jeweils überprüft werden sollen. Man kann dort eine von vier Voreinstellungen wählen: klein, mittel, groß und unbegrenzt.
Die Gruppierung „Speichern / Exportieren“ bietet die Möglichkeit, die Segmentierung als Volumen abzuspeichern indem man den gewünschten Namen in das gleichnamige Feld einträgt und auf „Segmentierung speichern“ klickt. Weiterhin öffnet sich mit dem Button „Standardsegmentierung öffnen“ der Dialog zur herkömmlichen Segmentierung mit den Daten aus der GPU-Segmentierung. Dort lässt sich die Segmentierung dann feinjustieren und anschließend mit dem neu hinzugefügten Button „GPU-Segmentierung öffnen“ wieder in die GPU-Segmentierung einladen.
Abbildung 59 zeigt beispielhaft zwei Ergebnisse der GPU-Segmentierung in der 3D Ansicht.

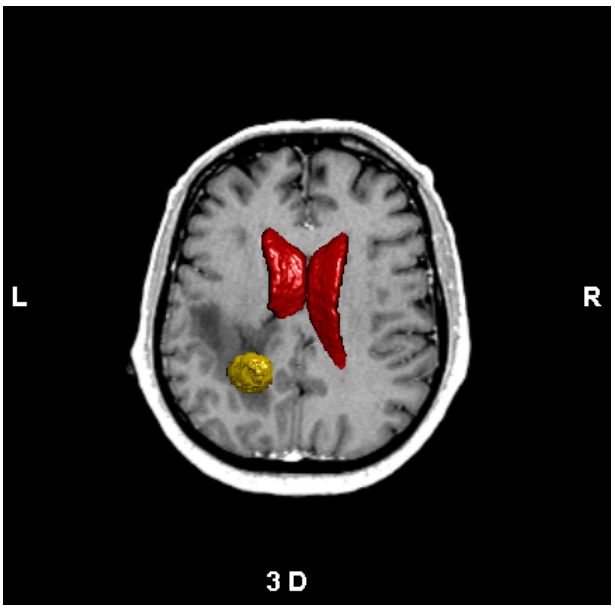


Abbildung 59: GPU-Segmentierung der Ventrikel (rot) und eines Tumors (gelb)
Quelle: eigene Darstellung (erstellt mit MOPS 3D)

6 Diskussion und Ausblick

Im Folgenden werden die Methodik und Ergebnisse hinsichtlich der Integration, der Performanz, der funktionalen Neuerungen und der Bedienbarkeit diskutiert. Im Anschluss werden weitere Entwicklungsmöglichkeiten vorgestellt.

6.1 Diskussion

Eines der Ziele bei der Integration der neuen Voreen Version war es, die Volume Rendering Komponente von MOPS 3D so zu gestalten, dass die zukünftige Integration neuer Versionen möglichst einfach ist. Problematisch bei dieser Aufgabe ist die Unvorhersehbarkeit der Änderungen, die an Voreen noch durchgeführt werden. Ursprünglich gestaltete sich die Volume Rendering Komponente bereits so, dass ein Upgrade auf die nächste Voreen Version nicht viel Aufwand bedeutete. Allerdings liegen zwischen den Versionen r13 und 2.6.1 fast 21 Monate und 10 Versionen, unter denen sich Meilensteine mit grundlegenden Neuerungen befinden (z.B. Version 2.0 [vgl. Vora]). Die Änderungen, die bei der Integration die größten Probleme bereiteten, sind in Kapitel 4.1 aufgeführt. Betrachtet man das Voreen Changelog [Vora], in dem die Änderungen jeder Version aufgelistet sind, so lässt sich erkennen, wie häufig grundlegende Änderungen durchgeführt wurden, wie z.B. Prozessoren zu ersetzen, Funktionalität zu ändern oder Shader Code neu zu gestalten. Aufgrund dieser Unvorhersehbarkeit der Änderungen ist es nahezu unmöglich, MOPS 3D für künftige Upgrades von Voreen zu wappnen. Es ist wahrscheinlich, dass die Integration von zukünftigen Voreen Versionen – zumindest für ein- oder zweistelligen Versionsnummern (z.B. 2.0 oder 2.5) – immer einen zusätzlichen Aufwand für den

Programmierer mit sich bringt, der im Vorfeld kaum abzuschätzen ist.

Die Arbeit mit Voreen gestaltete sich aus mehreren Gründen nicht immer als einfach. Zum einen sind die von Voreen verursachten OpenGL Fehlermeldungen oft nicht aussagekräftig. Außerdem ist die Programmierung der Shader mit GLSL schwierig, da es kaum geeignete Debugging Hilfsmittel gibt und im schlimmsten Fall dazu kommt, dass der Anzeigetreiber abstürzt. Zum anderen traten bei der Integration der neuen Voreen Version durch die vielen grundlegenden Änderungen unerwartete Probleme auf (siehe Kapitel 4.1). Beispielsweise wurde der Prozessor zum Verbinden mehrerer Eingangsbilder in ein Ausgabebild durch einen Prozessor ersetzt, der nur zwei Eingangsbilder verbinden kann. Dieses Problem konnte durch den Entwurf eines neuen Prozessors, der beliebig viele Eingangsbilder miteinander verbindet, in Rücksprache mit den Voreen Entwicklern gelöst werden.
Überdies wurde das Nachrichtensystem, über das man mit den Prozessoren kommunizieren konnte, komplett entfernt. Damit die Prozessoren und ihre Eigenschaften direkt angesprochen werden konnten, war eine Umstellung der gesamten Kommunikation erforderlich war.

Ein weiteres Ziel war die Verbesserung der Performanz der Volume Rendering Komponente. Diese hat sich – hauptsächlich durch die Integration der neuesten Voreen Version – durchweg verbessert (siehe Kapitel 5.1). Durch das Voreen Upgrade war ein leichter Geschwindigkeitsvorteil anzunehmen. Dieser überstieg jedoch – abhängig von den Qualitätseinstellungen – die Erwartungen teilweise bei weitem (siehe Abbildung 46). Des Weiteren konnte eine zusätzliche Performanzverbesserung bei der Darstellung multipler Segmente (bzw. bei deren Deaktivierung) erreicht werden (siehe Abbildung 47). Zusammenfassend lässt sich sagen, dass die Performanz im Vergleich zur früheren Version für die Operationsplanung ausreichend gut ist. Für die intraoperative Anwendung ist sie bei der Darstellung von mehreren Segmenten allerdings noch zu niedrig.

Weiterhin war es ein wichtiges Ziel, die Funktionalität des Clipping insofern zu erweitern, dass sie um neue Clippingstrukturen ergänzt werden kann. Es erwies sich hierbei als schwierig, ein geeignetes Verfahren für das Clipping neuer Strukturen zu entwerfen, da das herkömmliche Clippingverfahren für Quader nur für Objekte mit wenig Polygonen geeignet ist (siehe Kapitel 4.2.1). Zwar hätte man dieses Verfahren auch auf weitere Clippingstrukturen anwenden können, aber bei komplexeren Strukturen würde das zu niedriger Performanz und einer ungenauen Beschneidung führen – speziell bei dem beispielhaft eingeführten Ellipsoid. Stattdessen bietet sich der Ansatz an, die Voxel, die nicht dargestellt werden sollen, direkt im Fragment Shader des Raycasters zu ermitteln. Dies ist nicht nur einfach zu realisieren und flexibel zu erweitern, sondern hat gegenüber anderen Ansätzen auch den Vorteil, dass er auf dem vielfach schnelleren Grafikkartenprozessor ausgeführt wird. Deshalb wird es auch für das inverse Clipping genutzt. Die Performanz dieses Verfahrens stellte sich zwar als wesentlich langsamer heraus, verglichen mit dem herkömmlichen Clippingverfahren für Quader (siehe Abbildung 51). Allerdings liegt sie in einem vertretbaren Bereich, besonders wenn man bedenkt, dass die Testmaschine, auf der die Messungen stattfanden, eine mittelklassige Grafikkarte benutzt.
Des Weiteren sollte bei der Erweiterung des Clipping vor allem darauf geachtet werden, das Hinzufügen neuer Clippingstrukturen für den Programmierer möglichst einfach zu gestalten. Wie in Kapitel 4.2.1 beschrieben, befindet sich der Shader Code deshalb in einem eigenen Modul, das beliebig erweitert werden kann. Mit Hilfe der Oberflächenformel einer Struktur kann man den Shader einfach um eine Methode erweitern, die alles ausblendet, das sich außerhalb dieser Struktur befindet. Das zugehörige inverse Clipping lässt sich dann ähnlich leicht integrieren mit einer Methode, die gegenteilig dazu abfragt, ob sich der Voxel innerhalb der Struktur befindet.
Die Möglichkeit zur Modularisierung ist einer der Neuerungen der Voreen Version 2.6.1. Für das Hinzufügen neuer Clippingstrukturen stellte sie ein sehr hilfreiches Werkzeug dar, so dass der Programmierer in Zukunft die meisten benötigten Änderungen zentral an einer Stelle implementieren kann.

Ferner stand die Entwicklung einer Schnittstelle und Benutzeroberfläche für eine GPU-Segmentierung im Mittelpunkt der Arbeit. Beides sollte so konzipiert sein, dass es mit dem Framework zur GPU-Segmentierung funktioniert, welches Marius Wirths im Rahmen seiner Diplomarbeit [Wir12] entwickelte. Allerdings sollte es prinzipiell auch mit jedem anderen kompatiblen Framework für GPU-Segmentierung kommunizieren können.

Die Bedienbarkeit der Benutzeroberfläche orientiert sich dabei stark an dem Dialog zur herkömmlichen Segmentierung. Sie greift das Konzept von MOPS 3D auf, Strukturen direkt in den 2D Ansichten mit der Maus zu zeichnen und zu manipulieren (z.B. Freihandzeichnungen bei der herkömmlichen Segmentierung oder Clippingstrukturen bei der Volume Rendering Komponente) anstatt sie als komplizierte Steuerelemente im Dialog unterzubringen. Eine Evaluierung des Dialogs zeigt, dass er intuitiv bedient werden kann. Eine Ausnahme bildet hierbei die Umsetzung der Pinseldicke, die in Sachen Bedienbarkeit weiter verbessert werden sollte.

Bei der Evaluierung wurden beispielhaft MRT- und CT-Datensätze segmentiert, um den Nutzen der Benutzeroberfläche zu bewerten. Hierbei wird nicht die GPU-Segmentierung an sich evaluiert, da dies Gegenstand der Diplomarbeit von Marius Wirths [Wir12] ist. Es zeigt sich, dass sich mit dem Dialog sehr schnell und mit wenig Aufwand brauchbare Ergebnisse liefern lassen. Aufgrund der Echtzeitdarstellung in der 3D Ansicht lassen sich Fehler (d.h. ein Wachsen der Segmentierung über bestimmte Grenzen hinaus, die nicht überschritten werden sollen) früh bemerken und sehr schnell verhindern, indem man den Segmentierungsprozess anhält und mit dem Schieberegler zu der Iteration zurückspringt, an der die Region ausläuft. Unnötiger Overhead wird dadurch vermieden, dass man nicht von vorne segmentieren muss, sondern ab der ausgewählten Iteration fortfahren kann. Beides ist ein großer Vorteil im Vergleich zur herkömmlichen Segmentierung. Dort ist Region Growing aufgrund der Triangulierung nicht nur zeitaufwändiger, auch die Fehlersuche gestaltet sich als sehr schwierig, da einzelne Iterationen nicht einsehbar sind.

Im Vergleich zur herkömmlichen Segmentierung bietet der GPU-Segmentierungsdialog allerdings nicht so viele Möglichkeiten, da er auf Region Growing als Segmentierungsver-

fahren beschränkt ist. Diese Designentscheidung wurde getroffen, um den Dialog nicht zu überladen. Des Weiteren bieten nicht alle Funktionen der herkömmlichen Segmentierung einen Geschwindigkeitsvorteil, würden sie auf der GPU ausgeführt werden. Allerdings kann man das Segmentierungsergebnis beliebig von der GPU-Segmentierung in die herkömmliche Segmentierung exportieren und von dort wieder zurück importieren. Auf diese Weise lassen sich alle Funktionen der herkömmlichen Segmentierung mit denen der GPU-Segmentierung beliebig verknüpfen.

Insgesamt wurde die bestehende Volume Rendering Komponente in MOPS 3D in den Aspekten Performanz und Funktionalität weiterentwickelt in einen leistungsfähigen Bestandteil der Applikation, der der klinischen Anwendung von großem Nutzen sein kann. Durch die Integration der GPU-Segmentierung in MOPS 3D in Form einer Schnittstelle und passenden Benutzeroberfläche wurde eine schnelle und hilfreiche Art der Segmentierung hinzugefügt, die ihre Ergebnisse mit Volume Rendering in Echtzeit darstellt.

6.2 Ausblick

Da der Grundstein für die Verwendung von weiteren Clippingstrukturen in dieser Arbeit gelegt wurde, wäre es sinnvoll, dem beispielhaft integrierten Ellipsoid zusätzliche brauchbare Strukturen folgen zu lassen, wie z.B. einen Zylinder.
Ein großer Nachteil wäre dabei jedoch die fehlende Möglichkeit zur Rotation der Clippingstruktur. Möchte man mit dem Beispiel des Zylinders etwa ein Gefäß abbilden, so stößt man sehr schnell an die Grenzen, da der Zylinder immer eine feste Orientierung hat und sich nicht rotieren lässt. Hier wäre es angebracht, Rotation für Clippingobjekte zu implementieren, die sich ähnlich einfach mit der Maus steuern lässt wie das Verschieben und Skalieren von Clippingobjekten.
Eine zusätzliche Weiterentwicklung der Clippingstrukturen wäre denkbar, so dass sie an beliebiger Stelle mit der Maus verformt werden können. Am Beispiel des Zylinders würde das bedeuten, dass man durch Erweitern und Verengen an verschiedenen Stellen

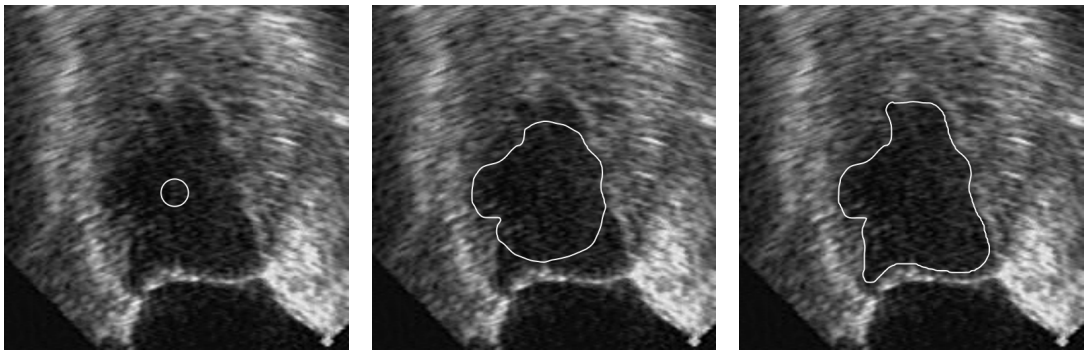
einen Zylinder formen kann, der der ungleichmäßigen Struktur eines Gefäßes entspricht. Weiterhin sollte die Darstellungsgeschwindigkeit der neuen Clippingstrukturen erhöht werden. Eine Möglichkeit wäre hierbei, Genauigkeit für Geschwindigkeit zu opfern, d.h. in der Praxis, dass z.B. nur jedes dritte Voxel überprüft wird und falls es außerhalb des Clippingbereichs liegt, werden automatisch alle benachbarten Voxel ebenfalls ausgeblendet.

Für die Volume Rendering Darstellung wäre es generell denkbar, das bestehende Modell der Transferfunktionen zu erweitern. Die Transferfunktion ist in MOPS 3D bisher derartig integriert, dass sie nur durch zwei Schwellwerte des Grauwertebereichs (sowie einer Farbe und einem Opazitätswert) beschrieben wird. Für die professionelle Anwendung wäre die Integration gradientenbasierter Transferfunktionen sinnvoll, mit deren Hilfe es möglich ist, Strukturgrenzen klarer zu definieren und Rauschen vorzubeugen.
Vorstellbar wäre auch das Erweitern der grauwertbasierten Transferfunktion um das freie Einzeichnen einzelner Punkte in das Grauwerthistogramm anstelle von Schwellwerten (ähnlich wie bei VoreenVE, siehe Abbildung 3). Das würde zwar möglicherweise eine komplexere Interaktion erfordern, gleichzeitig aber auch genauere Darstellungsergebnisse ermöglichen.

Hinsichtlich der GPU-Segmentierung sind weitere Verbesserungsmöglichkeiten denkbar. So hinkt speziell die Umsetzung der Pinseldicke nicht nur aufgrund ihrer Verständlichkeit, sondern ist auch unzureichend zum Eingrenzen von Strukturen geeignet. Sein primärer Nutzen ist das Verhindern des Auslaufens einer Segmentierung. Der Gedanke dahinter ist, dass man eine Lücke, durch die das Region Growing hindurch wächst, einfach mit einem Rechteck verschließt. Mit der Pinseldicke lassen sich solche Rechtecke zwar relativ gut erstellen, doch liegen diese aufgrund ihrer Architektur immer parallel zu einer der drei Ebenen (siehe Kapitel 4.4.3). Hier wäre es angebracht, diese Rechtecke frei rotierbar zu gestalten, so dass sie in der Lage sind, jede beliebige Lücke zu verschließen. Hier wäre es auch denkbar, zusätzliche Zeichnungsarten zur Begrenzung einzuführen,

wie z.B. solide Kugeln (die in den 2D Ansichten als Kreise dargestellt werden) oder geschlossene (zweidimensionale) Formen, die gefüllt sind und ebenfalls rotiert werden können.
Neben der Rotation wäre auch die Translation und Skalierung eine sinnvolle Neuerung für die Saatpunktzeichnungen. Dies würde z.B. die Notwendigkeit eliminieren, eine Zeichnung neu anfertigen zu müssen, wenn man feststellt, dass man sie nicht an der korrekten Position begonnen hat. Ideal wäre hierbei ein Interaktionsmodell, das dem professioneller Zeichenprogramme wie Adobe Photoshop gleicht, so dass Zeichnungen mit der Maus direkt verschoben oder mit den Pfeiltasten pixelgenau versetzt werden können.

Zusätzlich würde die Umsetzung der sog. *aktiven Konturen* (bzw. *Snakes*) eine sinnvolle Neuerung für die GPU-Segmentierung darstellen. Bei diesem Verfahren malt der Benutzer eine Zeichnung in den Datensatz (z.B. einen Kreis), welche dann über mehrere Iterationen hinweg wächst und sich der umliegenden Struktur (z.B. einem rundlicher Tumor) anpasst. Abbildung 60 verdeutlicht diesen Prozess.



(a) (b) (c)

Abbildung 60: Drei verschiedene Iterationen einer aktiven Kontur auf dem Ultraschallbild eines Herzen

Quelle: [XYJP00, S. 6]

Der Algorithmus für die aktiven Konturen wurde von Marius Wirths bereits prototypisch implementiert. Für die Zukunft wäre demnach eine Integration dieses Verfahrens in die GPU-Segmentierung empfehlenswert.

7 Anhang

Anhang A

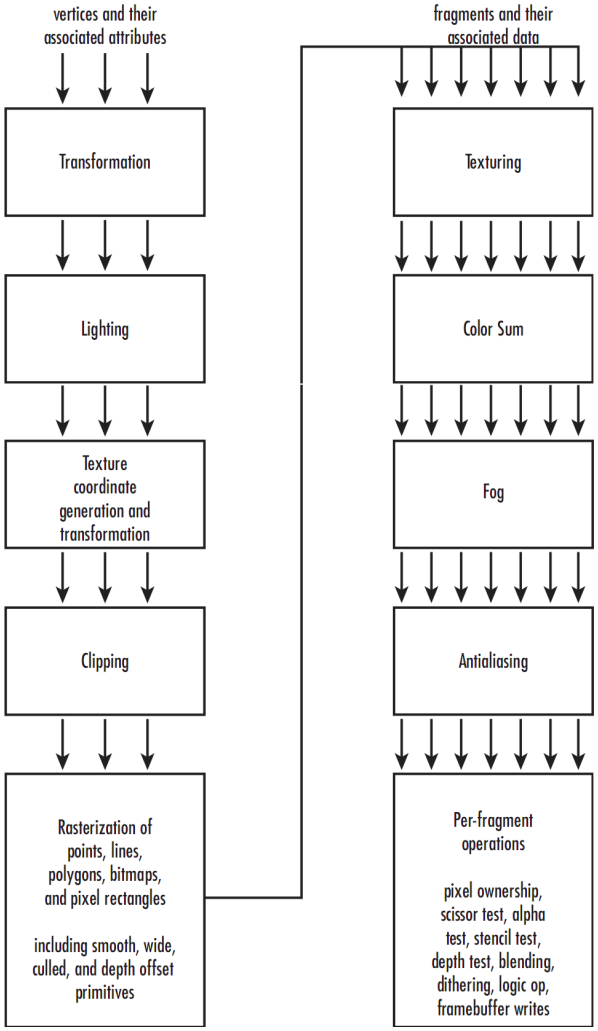


Abbildung 61: Detaillierte Darstellung der OpenGL Rendering Pipeline

Quelle: [WJLN07, S. 517]

Anhang B

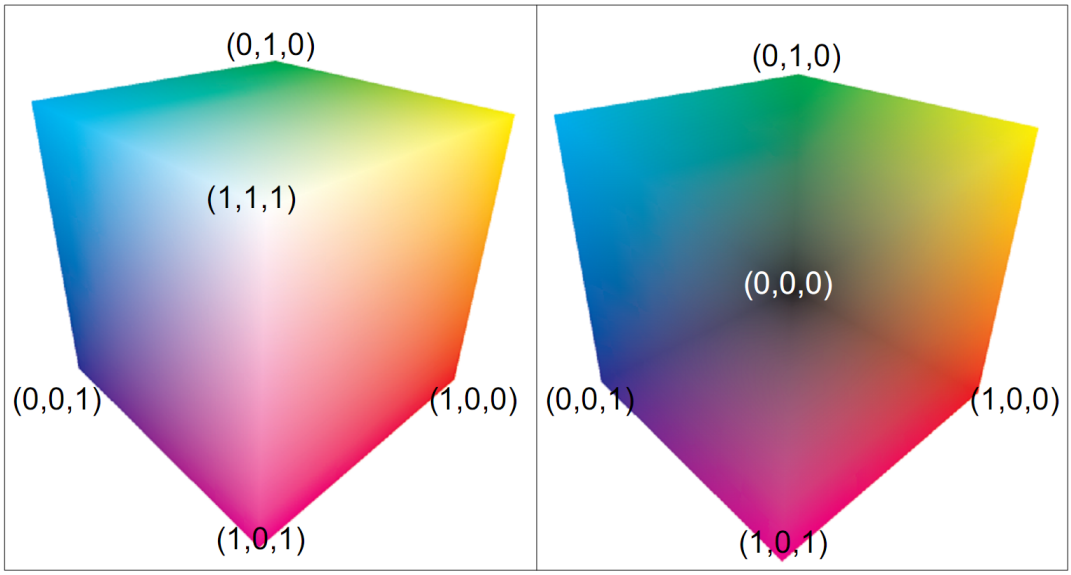


Abbildung 62: Ein- und Austrittstexturen des MeshEntryExitPoints Prozessors in Voreen

Quelle: [KW03, S. 3]

Anhang C

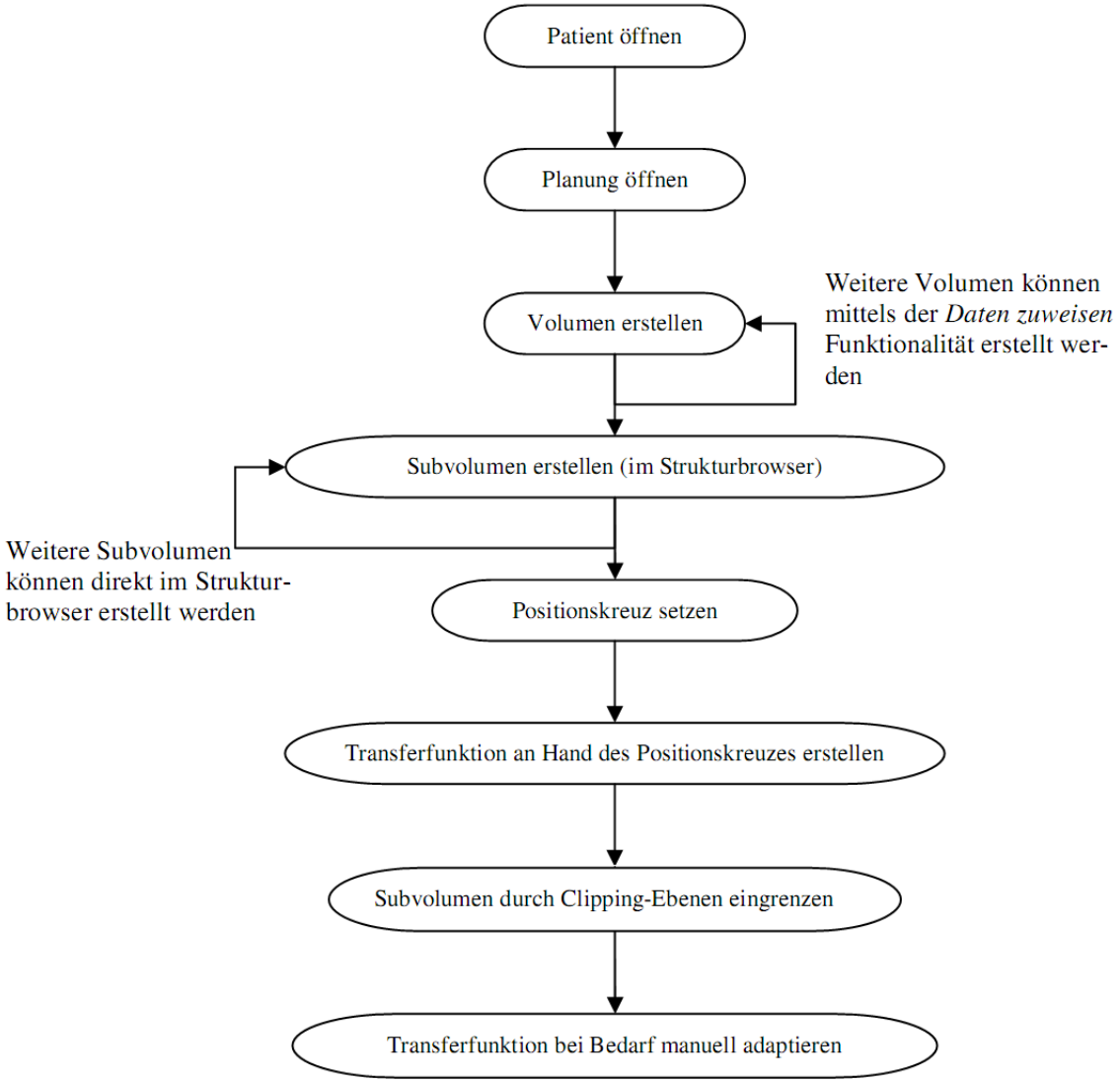


Abbildung 63: Ablauf der Nutzerinteraktion der Volume Rendering Komponente von MOPS 3D

Quelle: [Fre09, S. 57]

Anhang D

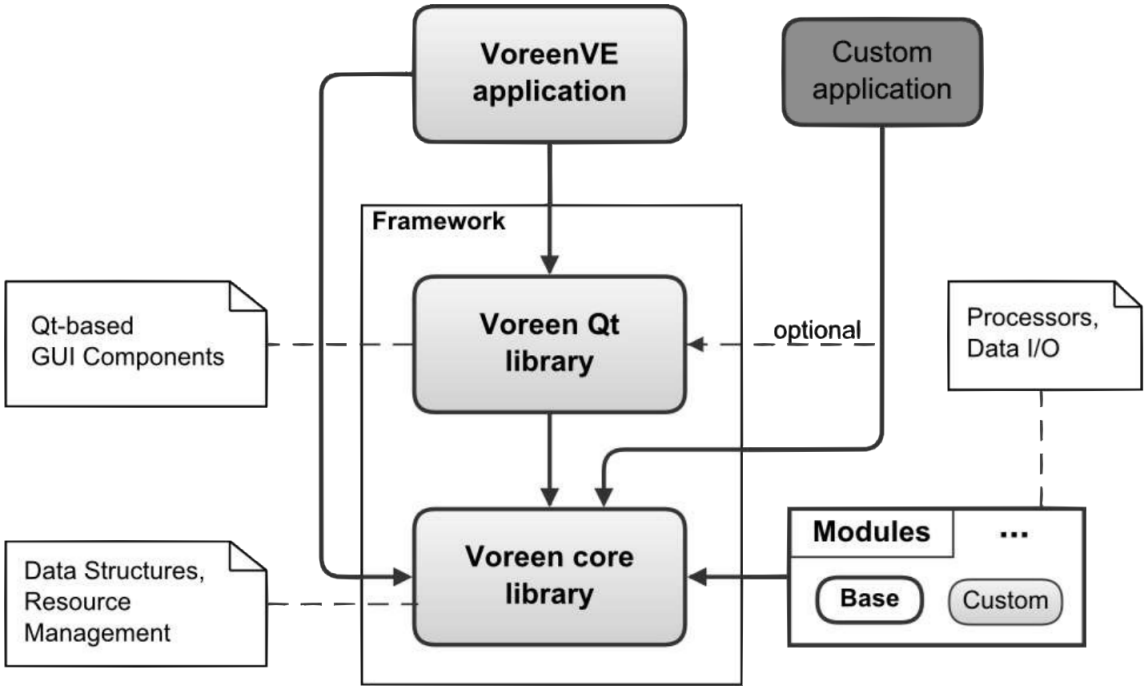


Abbildung 64: Architektur von Voreen

Quelle: [Vord, S. 26]

8 Abbildungsverzeichnis

9 Quellcodeverzeichnis

10 Literaturverzeichnis

[AB94] R. Adams und L. Bischof. „Seeded Region Growing“. In: *IEEE Transactions On Pattern Analysis and Machine Intelligence* 16 (6 1994), S. 641–647.

[Bir10] W. Birkfellner. *Applied Medical Image Processing: A Basic Course*. CRC Press, 2010.

[Eis+02] U. Eisenmann u. a. „Ein Computersystem zur Planung und Durchführung neurochirurgischer Interventionen“. In: *Biomedizinische Technik / Biomedical Engineering* 47 Nr. s1a (2002), S. 49–52.

[Fre09] A. Freudling. „Evaluation von GPU-basierten Verfahren zur Volumenvisualisierung und Integration in ein vorhandenes Operationsplanungssystem“. Diplomarbeit. Ruprecht-Karls-Universität Heidelberg / Hochschule Heilbronn, 2009.

[GWE04] R. C. Gonzalez, R. E. Woods und S. L. Eddins. *Digital Image Processing Using MATLAB*. Prentice Hall, 2004.

[KBR11] J. Kessenich, D. Baldwin und R. Rost. *The OpenGL Shading Language*. Version 4.20. [Online; Stand 17.10.2011]. Sep. 2011. URL: `http://www.opengl.org/registry/doc/GLSLangSpec.4.20.8.clean.pdf`.

[KW03] J. Krüger und R. Westermann. „Acceleration techniques for GPU-based volume rendering“. In: *Proceedings of IEEE Visualization VIS '03* (2003), S. 287–292.

[LC87] W. E. Lorensen und H. E. Cline. „Marching Cubes: A high resolution 3D surface construction algorithm“. In: *SIGGRAPH Comput. Graph.* 21 (4 1987), S. 163–169.

[LS92] L. Luqi und R. Steigerwald. „Rapid software prototyping“. In: *Proceedings of the TwentyFifth Hawaii International Conference on System Sciences* 2 (5 1992), S. 9–11.

[Met+06] R. Metzner u. a. „Pre- and Intraoparative Processing and Integration of Various Anatomical and Functional Data in Neurosurgery“. In: *Stud Health Technol Inform.* 124 (2006), S. 989–994.

[MS+09a] J. Meyer-Spradow u. a. *Rapid Prototyping of Volume Visualizations in Collaboration with Domain Experts*. Techn. Ber. 04-09-I. Institut für Informatik, Westfälische Wilhelms-Universität Münster, 2009.

[MS+09b] J. Meyer-Spradow u. a. „Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations“. In: *IEEE Computer Graphics and Applications* 29.6 (2009), S. 6–13.

[PB07] B. Preim und D. Bartz. *Visualization In Medicine*. Morgan Kaufman Publishers, 2007.

[RLK09] R. J. Rost und B. Licea Kane. *OpenGL Shading Language*. 3. Aufl. Addison-Wesley, 2009.

[SA10] M. Segal und K. Akeley. *The OpenGL Graphics System: A Specification*. Version 4.0. [Online; Stand 28. September 2011]. März 2010. URL: http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf.

[Sab88] P. Sabella. „A rendering algorithm for visualizing 3D scalar fields“. In: *SIGGRAPH Comput. Graph.* 22 (4 1988), S. 51–58.

[Vora] *Voreen Changelog*. [Online; Stand 21.11.2011]. Institut für Informatik, Westfälische Wilhelms-Universität Münster. URL: http://www.voreen.org/262-Changelog.html.

[Vorb] *Voreen Mailing List*. [Online; Stand 28.12.2011]. URL: http://article.gmane.org/gmane.comp.graphics.voreen.general/398.

[Vorc] *Voreen System Requirements*. [Online; Stand 25.11.2011]. Institut für Informatik, Westfälische Wilhelms-Universität Münster. URL: http://voreen.org/98-System-Requirements.html.

[Vord] *Voreen Tutorial Slides*. [Online; Stand 28.12.2011]. URL: http://voreen.org/files/voreen-tutorial-vis10.pdf.

[Wir12] M. Wirths. „Entwicklung einer visuell interaktiven Segmentierungskomponente unter Verwendung von OpenCL für das Operationsplanungssystem MOPS 3D“. Diplomarbeit. Ruprecht-Karls-Universität Heidelberg / Hochschule Heilbronn, 2012.

[WJLN07] R. S. Wright Jr., B. Lipchak und Haemel N. *OpenGL SuperBible*. 4. Aufl. Addison-Wesley, 2007.

[XYJP00] C. Xu, A. Yezzi Jr. und J. L. Prince. „On the Relationship between Parametric and Geometric Active Contours“. In: *Proceedings of 34th Asilomar Conference on Signals, Systems, and Computers* (2000), S. 483–489.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht.

Heidelberg, den

_____

Sascha Diatschuk